Closure Representations on the .NET CLR

Experiments and Observations

Don Syme,
Microsoft Research, Cambridge

A real problem…

- .NET language-level interop is largely solved:
  - e.g. objects, calling conventions, exceptions
  - e.g. C# can easily call VB
  - e.g. SML can easily catch C# exceptions
  - e.g. SML.NET can understand C# classes
- BUT:
  - Scheme: Untyped function values
  - Standard ML: Typed function values
  - Haskell: Typed function values, also delayed computations
  - OCaml: Typed function values
  - Funnel: Typed function values, also delayed computations
  - Mercury: Typed function values
  - …

A real problem...

- No interop over ANY of these constructs
  - A function value from one language cannot easily be used in another language
  - There are subtle and not-so-subtle differences between them.
  - Even if there isn't, compilers all encode them in different ways.
- The question: Can we do anything about this?

Overview

- Definitions and Aims
- A Range of Possible Standardized Encodings
- Experimental Results
- Conclusions & Ways Forward

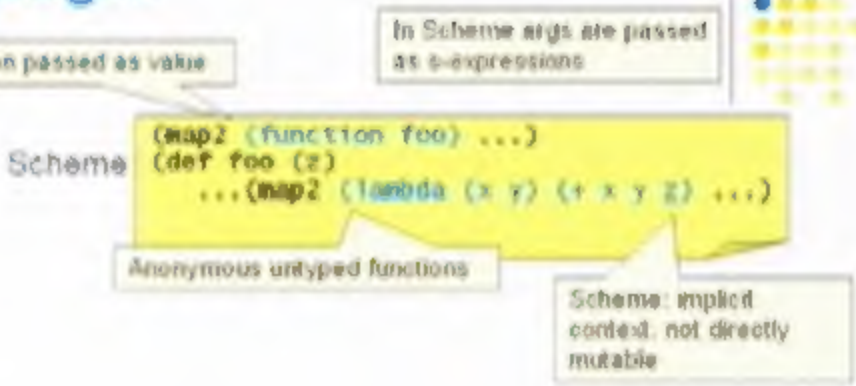
Source Languages

Source Languages

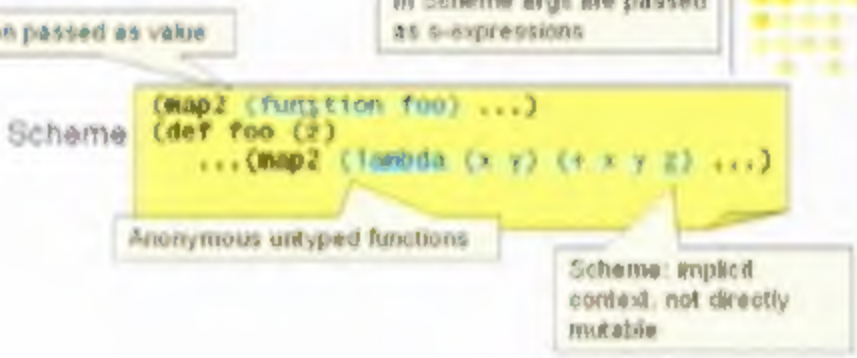
Scheme

```
(map2 (function foo) ...)
(def foo (z)
   ...(map2 (lambda (x y) (+ x y z)) ...)
```

Source Languages



Source Languages



Source Languages

In Scheme args are passed as s-expressions

Function passed as value

Scheme

```
(map2 (function foo) ...)
(def foo (z)
   ...(map2 (lambda (x y) (+ x y z)) ...))
```

Anonymous untyped functions

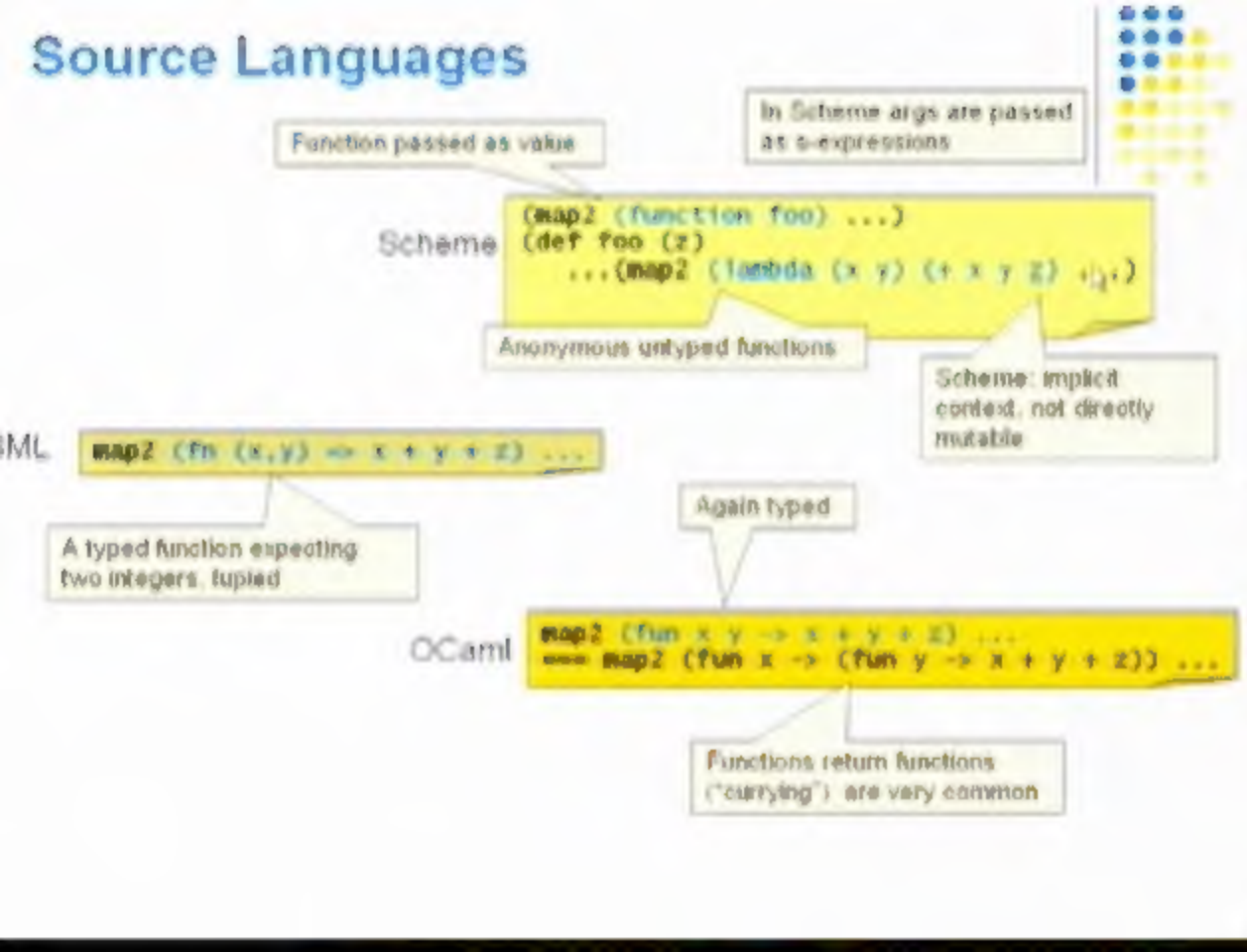
Scheme: implied context, not directly mutable

ML

```
map2 (fn (x,y) => x + y + z) ...
```

A typed function expecting two integers, tupled

Source Languages



Source Languages

In Scheme args are passed as expressions

```
(map2 ( + ( ) )
(def foo (z)
  (map2 ( ) ... ))
```

Anonymous untyped functions

Scheme implied context not directly relative

ML

```
map2 ( ) ( ) ( )
```

Again typed

A typed function expecting two integers tupled

```
map2 ( )
=== map2 (fun x -> (fun y -> x + y + z))
```

Haskell

```
map2 ( ) ( ) ( ) ( )
```

Functions return functions currying is very common

Again typed again lots of currying also lots of wrapped computations

Definitions

- Closures: functions in a context
- Closure expressions: declarations that specify new inner functions and their contexts (context is usually implicit)
- Function value: code pointer + data = an object allocated to hold pointer(s) to the function code plus the environment
- Function application: invoking a function value
- Function type: a general class of types that function values belong to.

Aims

- Fully-specified encoding of closures into MS-IL
- Easy to generate
- Supports separate compilation (no global analysis needed)
- Fast enough for e.g. OCaml
- Fits with MS-IL polymorphism/generics

Example

C#

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v.
```

Functional language

Example

C#

A function object is passed in

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map (fun y [illegible] New )

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v.
```

Functional programming

Example

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in

Here the same function object is [illegible]

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map [illegible]

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v.
```

Functional language

Example

C#

```
class Table<K,V> {
   void map (                    f )
   {
      for (int i = 0; ...) {

         val' = f(val)
      }
   }
   Pair<K,V>[] buckets,
}
```

A function object is passed in

Here the same function object is invoked in an Index

Here we pass in a function [illegible] call Table.map a new [illegible] allocated

```
let table = new Table<int,string>(10),
let rename = 0;

let process update (valOld,valNew) =
   table.map [illegible]

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v,
```

Functional language

Example

C#

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in

Here the same function object is invoked many times

Here we pass in a function [illegible] we call table.map a new closure will be allocated

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map [illegible]

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

[illegible]

The [illegible] elements from the environment Each time we call map a new function object will be allocated

Example

```
class Table<K,V> {
   void map ( ... f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in

Here the same function object is invoked many times

Here we pass in a function to be called. Table.map is now a new closure will be allocated

```
let table = new Table<int,string>(10);
let rename = 0;

let process update (valOld,valNew) =
   table.map ...

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

This closure expression uses elements from the environment. Each time we call map, a new function object will be allocated

When invoked the function object calls this function. It can have side-effects

Example

C#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {

         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map [illegible] .

let [illegible]

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Example

C#

```
class Table<K,V> {
   void map (                    f )
   {
      for (int i = 0,      ) {

         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in

There is an implied generic function type

Here the same function object is invoked many times

Here we pass in a function
Each time we call Table.map a new closure will be allocated

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map                                    New .

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v.
```

When invoked the function object calls the function. It can have side effects

The compiler captures values from the environment
Each time we call Map a new function object will be allocated

Example

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {

         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   Table.map [illegible] .

[illegible]

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Functional programming

Example

C#

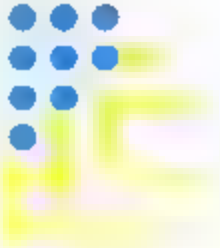
```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {

         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map
```

```
let
```

```
let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Part 2: Implementation Choices



Example

C#

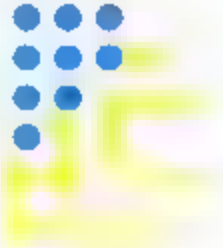
```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map [illegible] .

[illegible]

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

[illegible]

[illegible] expressions are [illegible] with the environment [illegible]

Part 2: Implementation Choices



Overview

- Note: for typed functional languages we really assume Generics have been added to the CLR

Generics == Parametric Polymorphism == Templates

Choices & Non-choices

```
class Table<K,V> {
   void map ( . . . . . f )
   {
      for (int i = 0; ...) {

         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]).

let clo[valOld,valNew](y) = rename (valOld,valNew,y).

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Choices & Non-choices

Non-choice function types must be structural

```
class Table<K,V> {
   void map ( [illegible] )
   {
      for (int i = 0; ...) {
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table map (new clo[valOld,valNew]).

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++,
      valNew
   else v
```

Choices & Non-choices

Non-choice: function types must be structural

```
class Table<K,V> {
   void map (                    f )
   {
      for (int i = 0; ...) {

         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

Encoding function types as

[illegible] structural equivalence

Note [illegible] make delegates structural

Note [illegible] still be [illegible]

```
let table = new Table<int,string>(10);
let rename = 0;

let process update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Choices & Non-choices

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

[illegible] functions objects are [illegible]

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map ([illegible] valNew).

let [illegible] valNew (y) = rename (valOld,valNew,y).

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Choices & Non-choices

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

Non-mutable functions: side-effects are [illegible]

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map (new [illegible] v -> new).

let clo[illegible] New (y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++,
      valNew
   else v
```

Choices & Non-choices

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

[illegible] functions/objects are [illegible]

```
let table = new Table<int,string>(10).
let rename = 0.

let process update (valOld,valNew) =
   table.map ([illegible]).

let clo[illegible]New (y) = rename (valOld,valNew,y)

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++,
      valNew
   else v
```

Choice: How is the environment stored w.r.t. the function object?

The relevant parts of the environment must be somehow accessed via fields of the function object

Choices & Non-choices

```
class Table<K,V> {
   void map ( [illegible] )
   {
      for (int i = 0; ...) {
         val' = f(val)
      }
   }
   Pair<K,V>[] buckets;
}
```

Non-choice fu
heap-allocate

```
let table = new Table<int,string>(10);
let rename = 0;

let process update (valOld,valNew) =
   table.map ( [illegible] );

let clo[illegible] (y) = rename (valOld,valNew,y)

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

[illegible] How to get from the function value to the [illegible] where the code is stored in a closed form function object

[illegible] setting from code B is an indirect call in the .NET CLR. Class tables, interface tables, delegates, code pointers are the only choices

[illegible] How is the environment stored w.r.t. the function object?

The relevant parts of the environment must be somehow accessed as fields of the function object

The Simplest Implementation

- Encoding into virtual methods

```
abstract class System.Func<A,B> {
    virtual B apply(A);
}

class clo : System.Func<string,string> {
   private string valOld;
   private string valNew
   override string apply(string) {     }
}

      new clo(...)
      F.apply()
```

- Code = virtual method
- Environment stored inline in object

The Simplest Implementation

- Encoding into virtual methods

This is the space of function types

```
abstract class System.Func<A,B> {
    virtual B Apply(A);
}

class Clo : System.Func<string,string> {
   private string valOld;
   private string valNew
   override string Apply(string) { ... }
}

      new Clo(...)
      f.Apply() ...
```

Each closure becomes a delegee class

- Code = virtual method
- Environment stored inline in object

The Simplest Implementation

- Encoding into virtual methods

This is the space of function types

```
abstract class System Func<A,B> {
    virtual B apply(A);
}

class clo : System Func<string,string> {
   private string valold;
   private string valNew
   override string apply(string) {     }
}

     new clo(...)
   ; f.apply() ...
```

Each closure becomes a delegee class

Closure construction makes an object

Application becomes virtual method invocation

- Code = virtual method
- Environment stored inline in object

The Delegate Implementation

- Encoding into virtual methods

```
delegate B System.Func<A,B>(A)

class clo {
   private string valOld,
   private string valNew,
   string apply(string) {      }
}


      new clo(...)
      new System.Func<string,string>(clo.apply)

      f.Invoke("abc");
```

- Code = method on delegee
- Environment stored in delegee

The Delegate Implementation

- Encoding into virtual methods

This is the space of function types

```
delegate B System.Func<A B>(A)

class clo {
   private string valOld,
   private string valNew,
   string apply(string) {      }
}

      new clo(...)
      new System.Func<string,string>(clo.apply)

      f.Invoke("abc");
```

Each closure becomes a delegee class

- Code = method on delegee
- Environment stored in delegee

The Delegate Implementation

- Encoding into virtual methods

This is the space of function types

```
delegate B System.Func<A,B>(A);

class clo {
   private string valOld;
   private string valNew;
   string apply(string) {      }
}

        new clo(...)
        new System.Func<string,string>(clo.apply)

        f.Invoke("abc");
```

Each closure becomes a delegee class

Closure construction makes a delegee and a delegate

- Code = method on delegee
- Environment stored in delegee

The Delegate Implementation

- Encoding into virtual methods

```
delegate B System.Func<A,B>(A)

class clo {
   private string valOld;
   private string valNew;
   string apply(string) {     }
}

      new clo(...)
      new System.Func<string,string>(clo.apply)

      f.Invoke("abc");
```

This is the space of function types

Each closure becomes a delegee class

Closure construction makes a delegee and a delegate

Application becomes delegate invocation

- Code = method on delegee
- Environment stored in delegee

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template" classes

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code),
   public E;
}
```

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template classes

This is the class of function types

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ . . . }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code),
   public E;
}
```

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template" classes

This is the class of function types

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Closure<E,A,B> : System.Func<A,B> {
   E env; method B *(E env,A x) code;
   public E;
}
```

This is a helper method for application

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template" classes

This is the class of function types

```
abstract class System.Func<A,B> {
   void * code;
}
```

This is a helper method for application

```
static method B app<A,B>(System.Func<A,B>, A x)
{ ... }
```

This class is a template for all closures

```
class System.Clo<E,A,B>  : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template classes

This is the class of function types

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo1<E,A,B>  : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code),
   public E;
}
```

This is a helper method for application

This class is a template for all closures

This is a C-function pointer type in MS-IL

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template classes

This is the class of function types

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B>  : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

This is a helper method for application

This class is a template for all closures

This is the environment

This is a C-function pointer type in MS-IL

A Library Implementation

- Encode into:
  - MS-IL C-style function pointers
  - Closure "template" classes

This is the class of function types

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

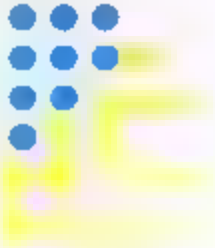
This is a helper method for application.

This class is a template for all closures

This is the environment

This is a C-function pointer type in MS-IL

Part 3: Further Details of Implementation Choices



Choices: Environments

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0, [illegible]) {
         val' = [illegible]
      }
   }
   Pair<K,V>[] buckets;
}
```

[illegible]

Choices: Environments

```
class Table<K,V> {
   void map (                    f )
   {
      for (int i = 0,      ) {

         val' = f(
      }
   }
   Pair<K,V>[] buckets,
}
```

Choices: Environments

```
class Table<K,V> {
   void map (                    f )
   {
      for (int i = 0,          ) {

         val' = f
      }
   }
   Pair<K,V>[] buckets,
}
```

For closure templates, a flat
effect ar e a ieved by
using tupling e g
instantiating 'E' with e g
Pair int int> or
Tuple4<int int int int

Flat via tupling

Always split

Full fu class

In our tests this s a new class
per closure when needed

Choices: Code

```
class Table<K,V> {
   void map (                    f )
   {
      for (int i = 0,      ) {

         val' = f
      }
   }
   Pair<K,V>[] buckets,
}
```

Choices: Code

```
class Table<K,V> {
   void map ( system.func<v,v  ? )
   {
      for (int i = 0,     ) {

         val' = i
      }
   }
   Pair<K,V>[] buckets,
}
```

Choices: Code

```
class Table<K,V> {
   void map ( system func<V      ? )
   {
      for (int i = 0,      ) {

         val' = !<

      }
   }
   Pair<K,V>[] buckets,
}
```

1 Vtable

Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V   f )
   {
      for (int i = 0      ) {

         val' = f
      }
   }
   Pair<K,V>[] buckets;
}
```

2 Delegates

Choices: Code

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0, [illegible]) {
         val' = [illegible]
      }
   }
   Pair<K,V>[] buckets,
}
```

Code Pointers Own

Choices: Code

```
class Table<K,V> {
   void map ( [illegible] f )
   {
      for (int i = 0, [illegible] ) {
         val' = f([illegible])
      }
   }
   Pair<K,V>[] buckets,
}
```

1 Vtable

[illegible] delegates

} Code Pointers Own

[illegible] Templates

P, defining [illegible] template
[illegible] and environment layout

Choices: Multiple Entries

Scheme

```
(map2 (lambda (x y) (+ x y z))

(def map2 (f l1 l2)
      (f (car l1) (car l2)))
```

```
(def add2 (f x l2) .
      (f x 4)
```

Choices: Multiple Entries

Scheme

```
(map2 (lambda (x y) (+ x y z))

(def map2 (f l1 l2)
      (f (car l1) (car l2)) .
```

```
(def add2 (f x l2) . .
      (f x 4)
```

Choices: Multiple Entries

Scheme

```
(map2 (lambda (x y) (+ x y z))

(def map2 (f l1 l2)
      (f (car l1) (car l2)) .
```

```
(def add2 (f x l2) .
      (f x 4)
```

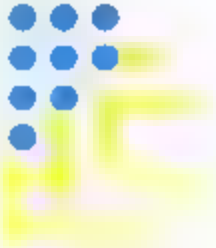
Choices: Multiple Entries

Scheme

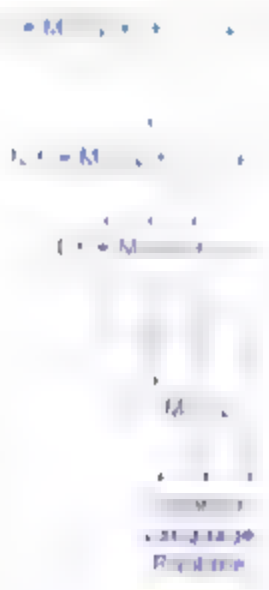
```
(map2 (lambda (x y) (+ x y z))

(def map2 (f l1 l2)
      (f (car l1) (car l2)) .
```

```
(def add2 (f x l2) .
      (f x 4)
```

Part 4: Experimental Results



ILX Implementations by Translation



Simulated Performance Load

- [illegible] program

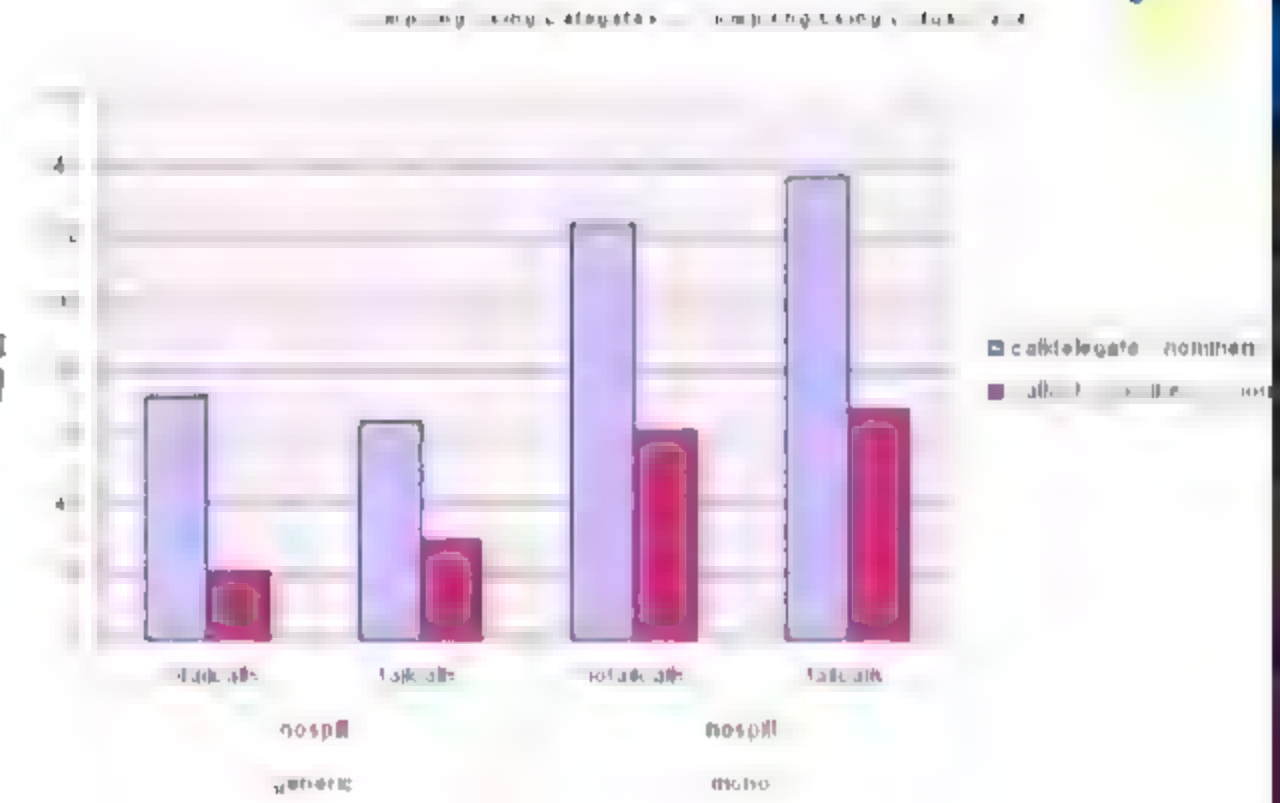
  [illegible]

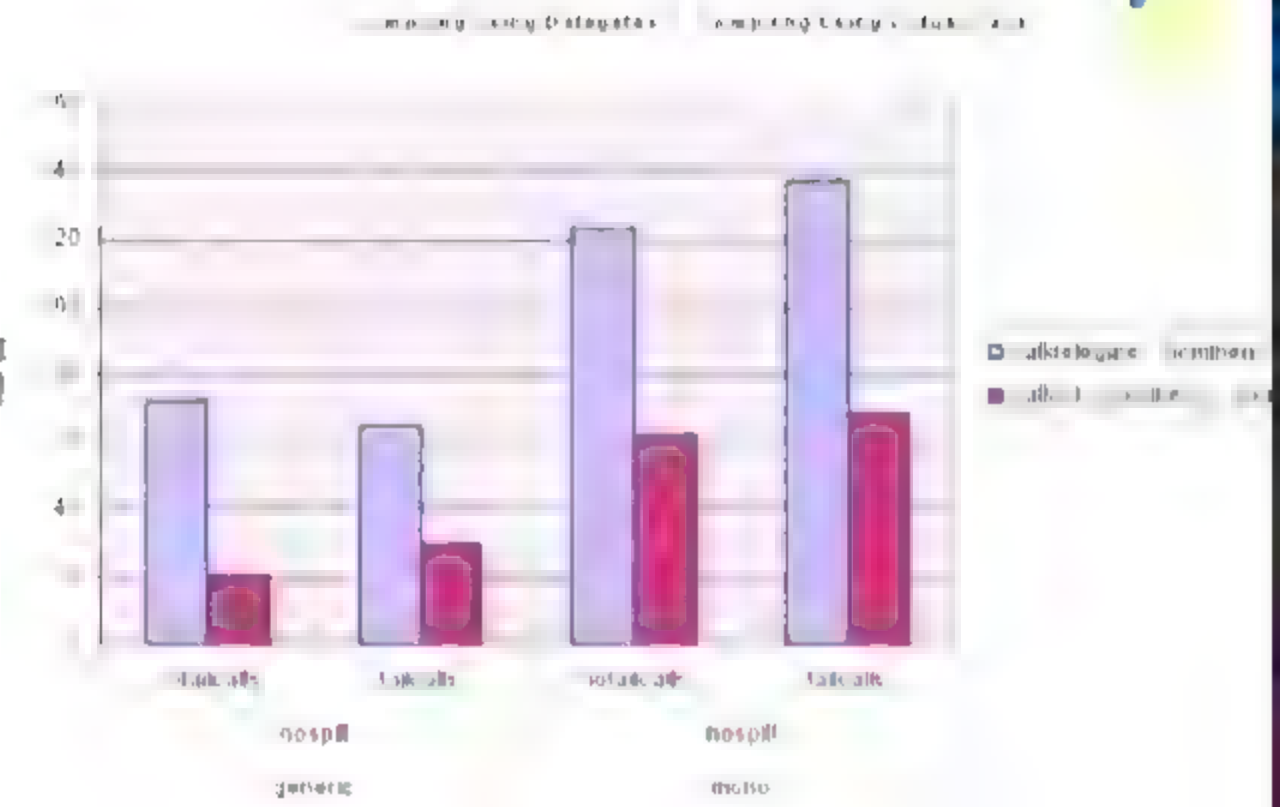
  All [illegible]

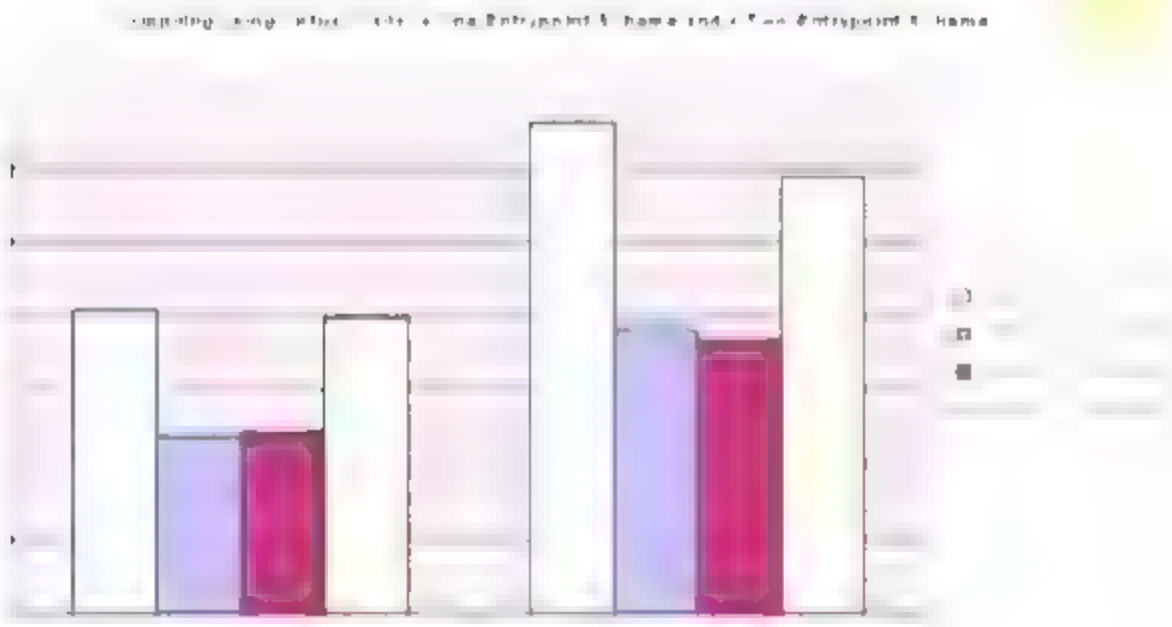
  [illegible]

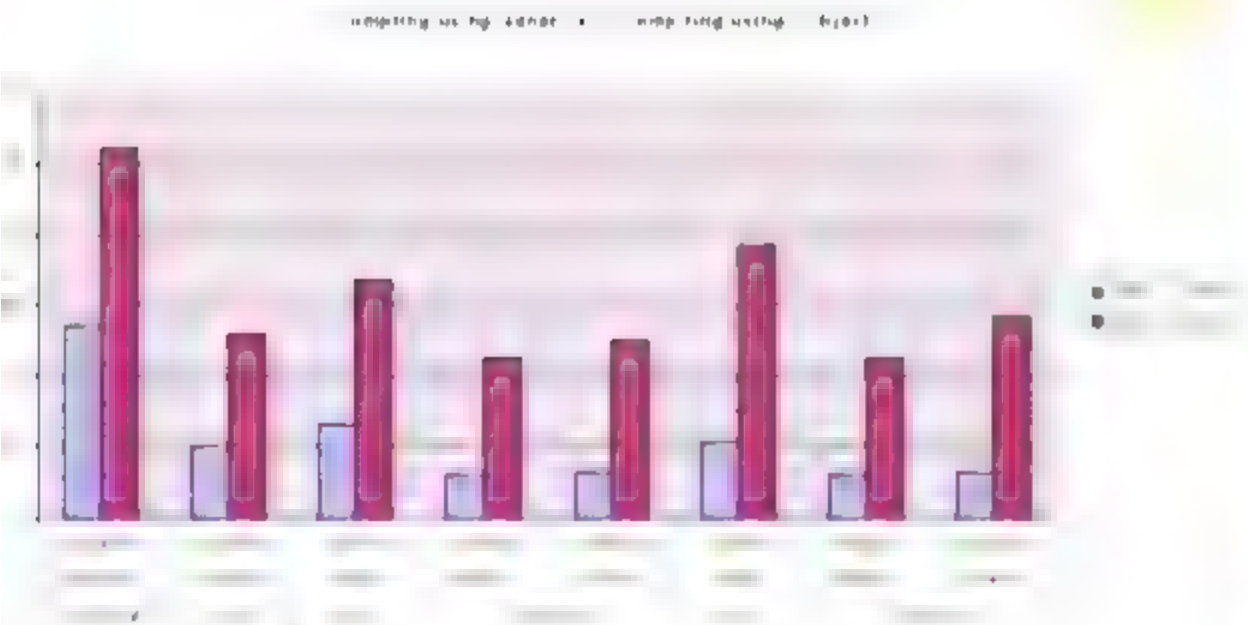
  [illegible]

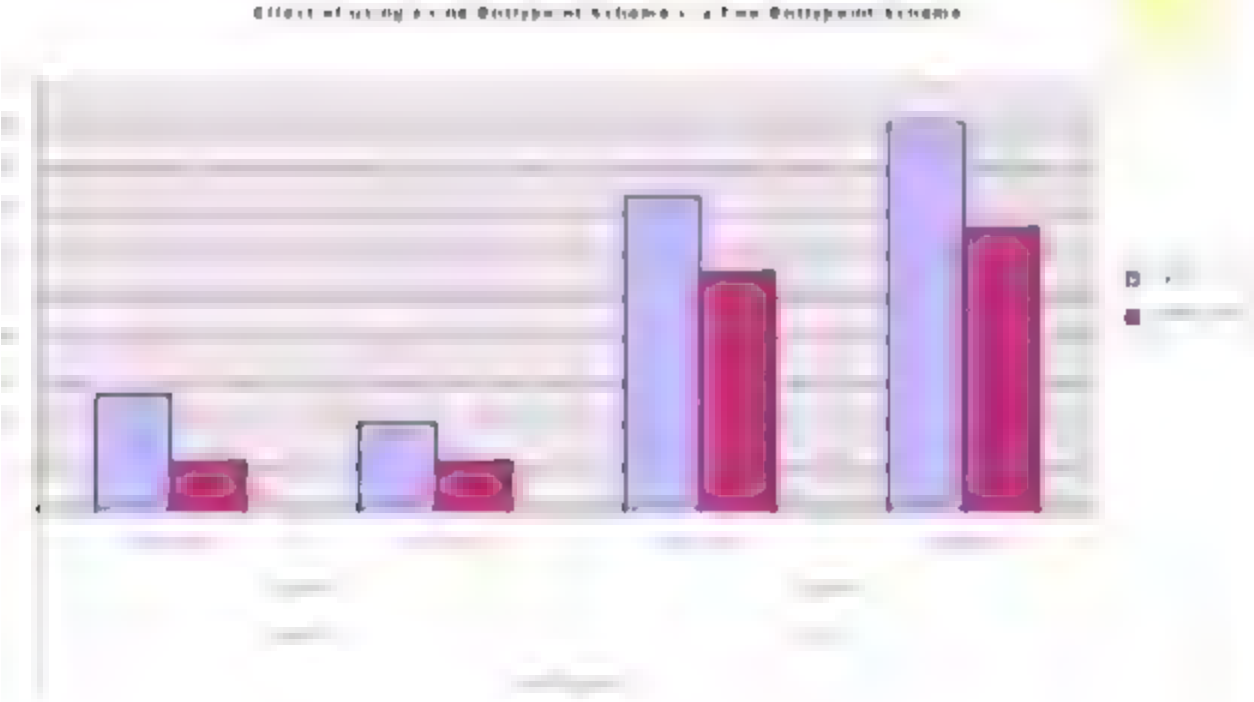
- BEST TIME =~ 11s

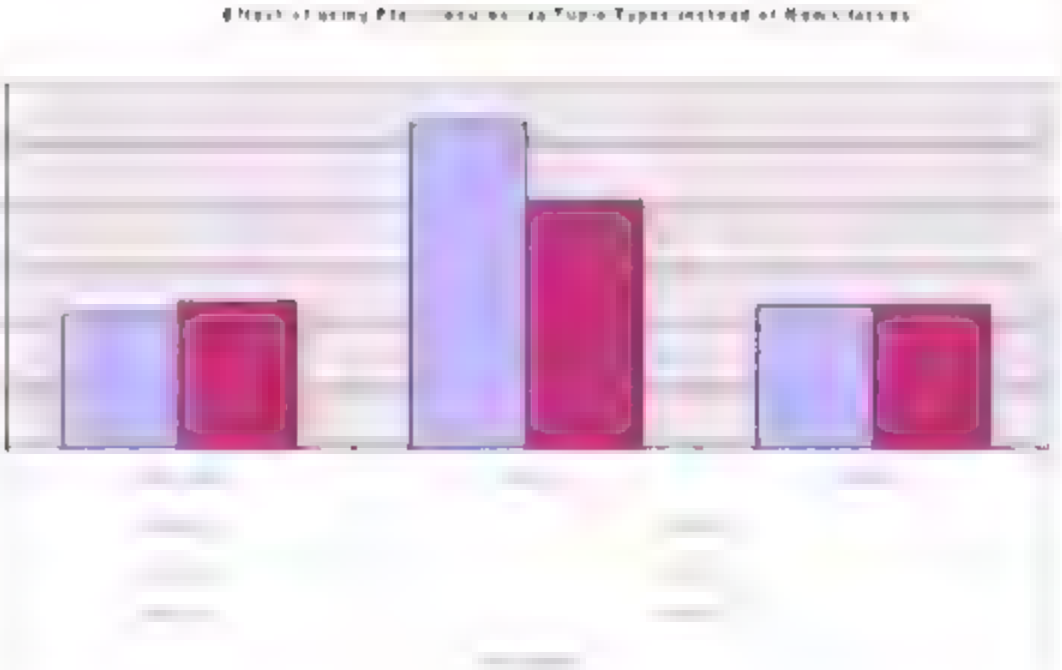


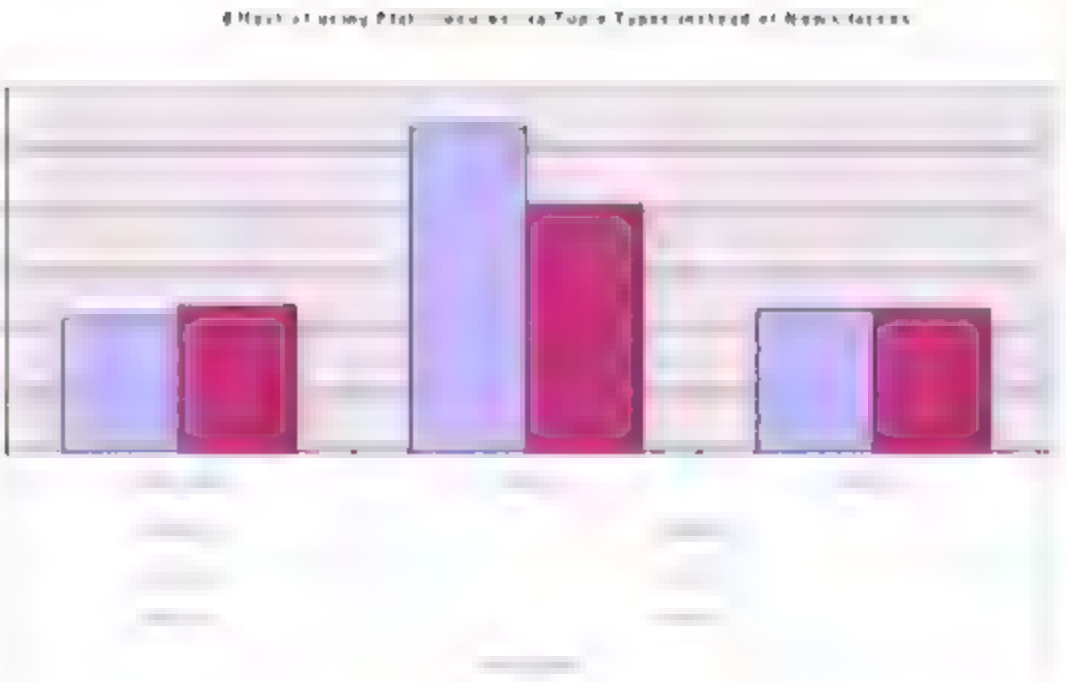


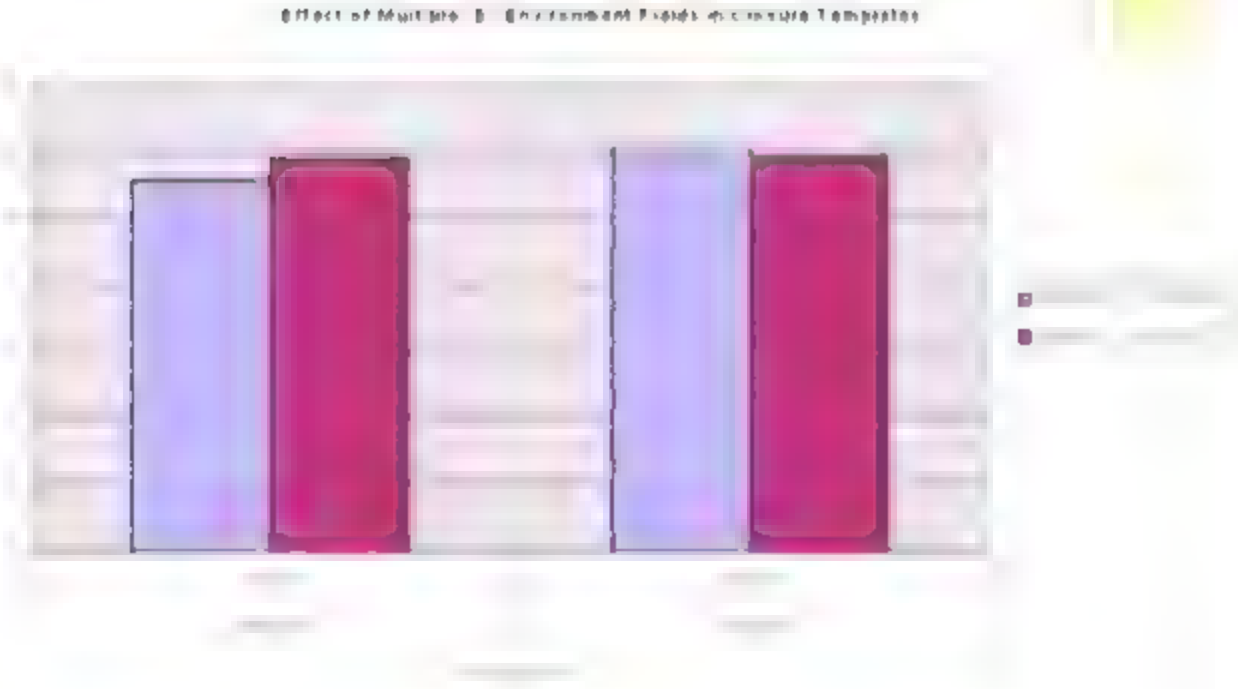


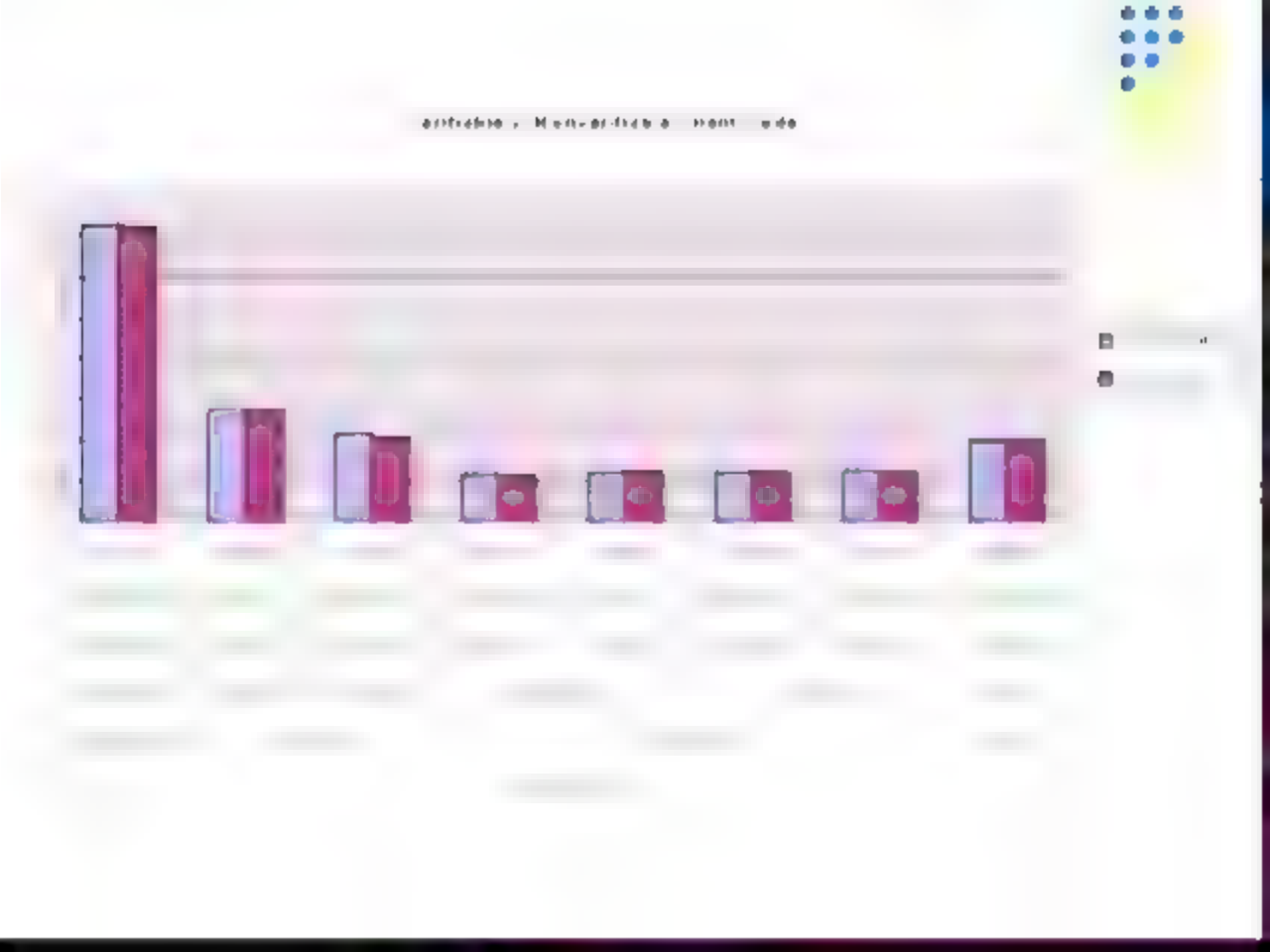


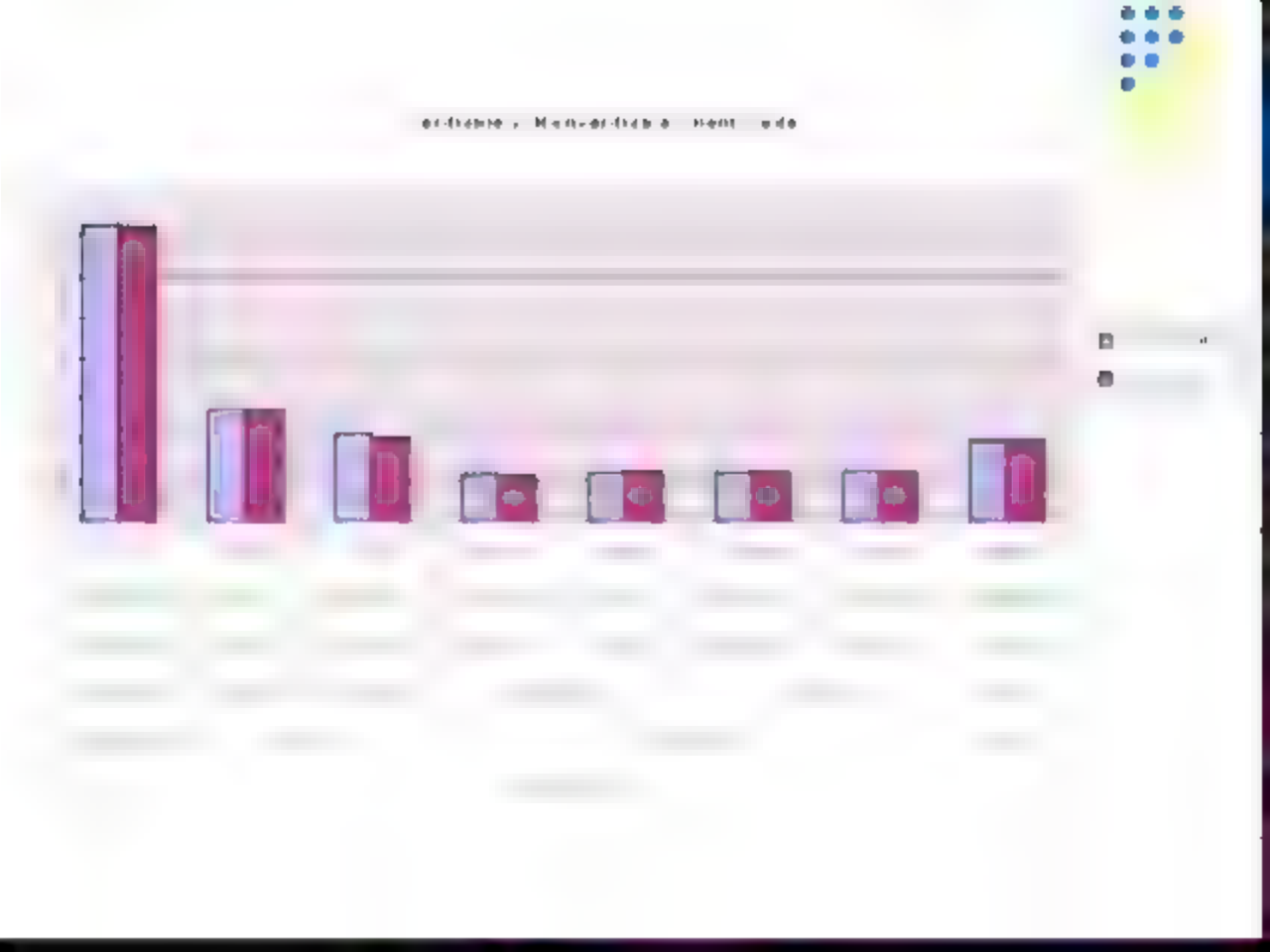












Conclusions

1. V1 Delegates suck
2. V1 Tailcalls suck
3. Generics are an extremely useful compilation device
4. Multiple entry points are feasible and worth it
5. Flat closures are feasible and worth it
6. Multiple environment slots are not worth it

7. A library-based approach is feasible

Conclusions

1. V1 Delegates suck
2. V1 Tailcalls suck
3. Generics are an extremely useful compilation device
4. Multiple entry points are feasible and worth it
5. Flat closures are feasible and worth it
6. Multiple environment slots are not worth it

7. A library-based approach is feasible

Related Concepts

- Classes/Object
  - Context is explicit: the fields
  - Can [illegible] multiple methods, static methods, reflection
- [illegible]
  - Context is explicit
  - [illegible]
- C# Delegates
  - Context is explicit
  - Code and data are tied
  - Also [illegible]
- Method pointer
  - [illegible]
  - Unboxed Delegate
    - Full [illegible]

Looking Forward

Ways ahead

- The principle, a collegiate performance of [illegible] plus a range of [illegible]

  - delegate

    .1 delegates

    +

    Event performance

    +

    [illegible] output

    +

    Environments alone in delegate

    +

    Multiple events

Summary

Problems

1. [illegible]

2. [illegible]

Future research

1. [illegible]

2. [illegible]

Questions?

Runtime Verification and .NET

Michael Barnett
Wolfram Schulte
Foundations of Software Engineering

Testing and Verification?

"Computer Science is akin to astronomy and astrology but it lacks the precision of the former and the popularity of the latter"

Specifications are Good...

Specifications are Good ..

- So why doesn't anyone write them?

Specifications are Good...

- So why doesn't anyone write them?
  - Limited usefulness dead documents

Specifications are Good ..

- So why doesn't anyone write them?
  - Limited usefulness dead documents
- But some people **do** write them

Specifications are Good...

- So why doesn't anyone write them?
  - Limited usefulness dead documents
- But some people **do** write them
  - Programmers write assertions

Specifications are Good ..

- So why doesn't anyone write them?
  - Limited usefulness dead documents
- But some people **do** write them
  - Programmers write assertions
  - Programmers write error checking

Specifications are Good...

- So why doesn't anyone write them?
  - Limited usefulness dead documents
- But some people **do** write them
  - Programmers write assertions
  - Programmers write error checking
  - Programmers write exception handling

Specifications are Good ..

- So why doesn't anyone write them?
  - Limited usefulness dead documents
- But some people **do** write them
  - Programmers write assertions
  - Programmers write error checking
  - Programmers write exception handling
  - Programmers write code!

It's the level, stupid. .

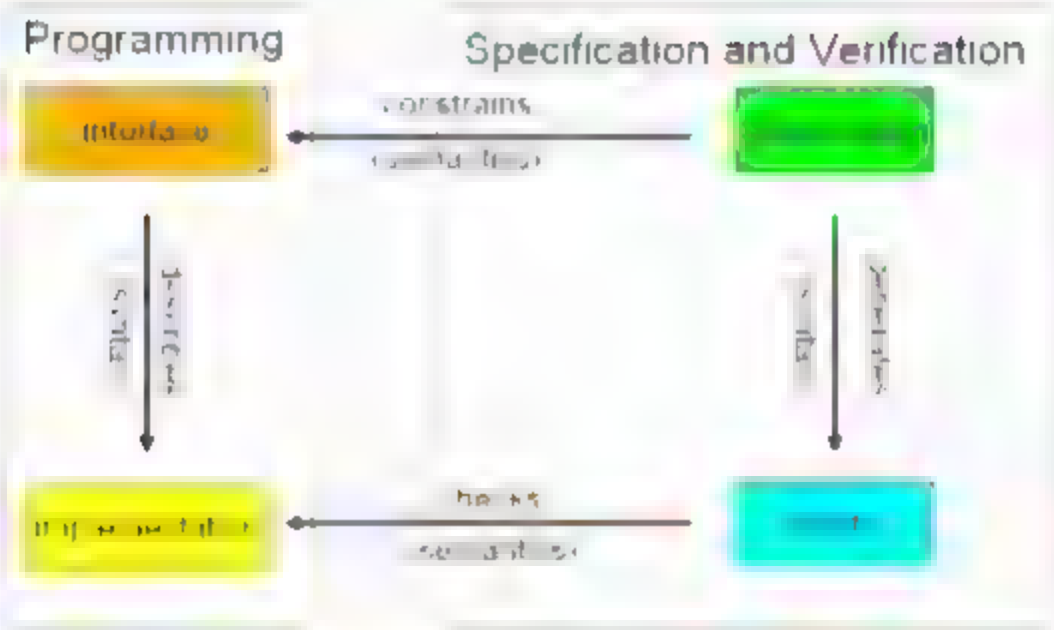
- Specifications would be more useful if they were:
  - At a higher level
  - Available separate from source code
  - Enforceable

It's the level, stupid. .

- Specifications would be more useful if they were:
  - At a higher level
  - Available separate from source code
  - Enforceable

i e., executable!

The Big Picture



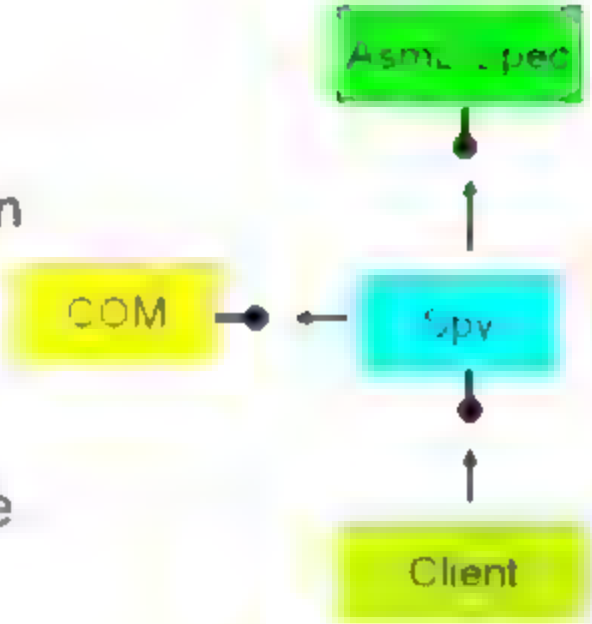
Previously

- Observational equivalence
- No instrumentation required

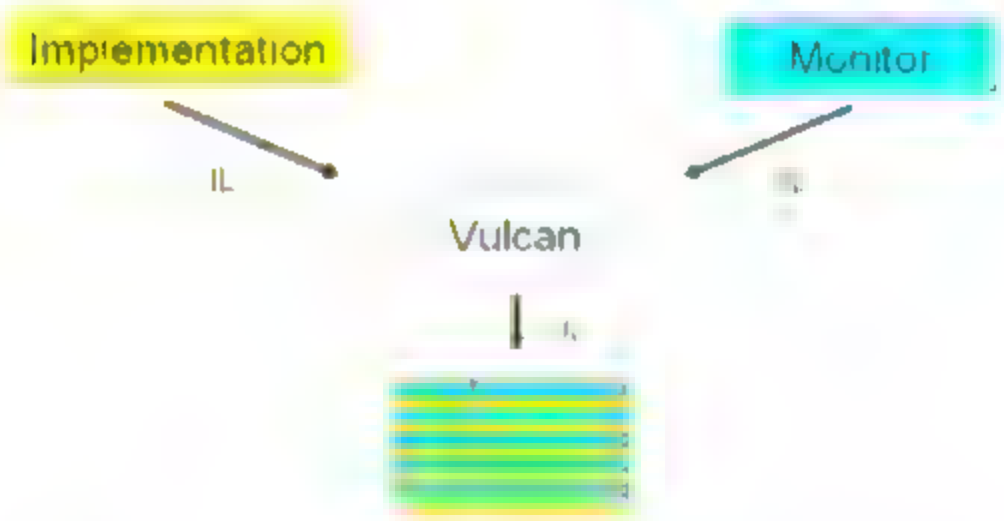
Non-determinism

Callbacks

State equivalence



Tighter Integration: NET



(Trivial) Example

```
interface IMath {
  double sqrt(double x)

}
```

```
class CMath : IMath {
  double sqrt(double x)
    [ ]
    .
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    [ x >= 0; ]
  bool sqrt_post(double x)
    { result * result == x }
}
```

Monitored Implementation

```
class CMath {

  double sqrt(double x)
    { double result
      assert(x >= 0);
      modified body
      assert(result*result == x);
      return result
    }
}
```

Monitor Language

```
class I_Contract {
  bool f_pre( ) { }
  bool f_post( ) { }
  bool f_classInvariant( ) { }
  Object f_exception( ) { }
}
```

Monitor Language

```
class F_Contract {
  bool f_pre( ) { }
  bool f_post( ) { }
  bool f_classInvariant( ) { }
  Object f_exception( ) { }
}
```

What to specify?

- Properties
  - Pre-/post-conditions
- Calling protocols
  - Required method calls *mandatory calls*
- Model State
  - Required persistence *objects*
- Exceptions

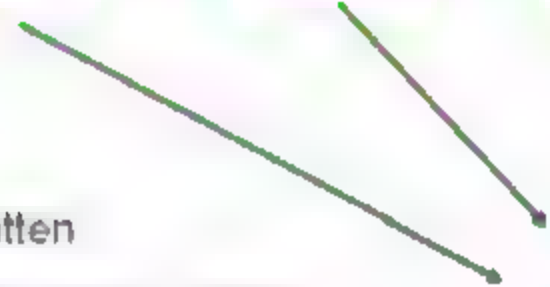
(Trivial) Example

```
interface IMath {
  double sqrt(double x);
}
```

```
class CMath : IMath {
  double sqrt(double x)
    [ ... ]
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    [ x >= 0; ]
  bool sqrt_post(double x)
    { result * result == x }
}
```

Monitors



- Hand-written
- Translated from a higher-level specification

Monitor

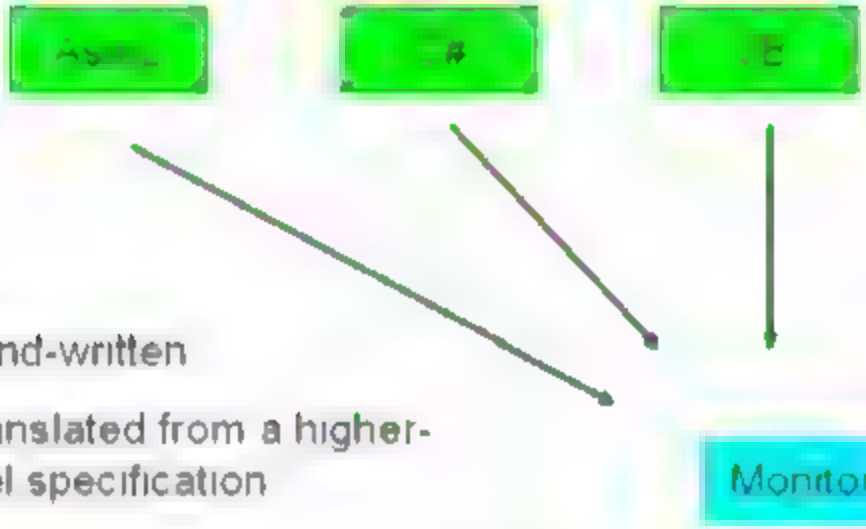
(Trivial) Example

```
interface IMath {
  double sqrt(double x);

}
```

```
class CMath : IMath {
  double sqrt(double x)
    [ ... ]

}
```

```
class IMath_Contract : CMath {

  bool sqrt_pre(double x)
    [ x >= 0; ]

  bool sqrt_post(double x)
    { result * result == x }
}
```

Monitors



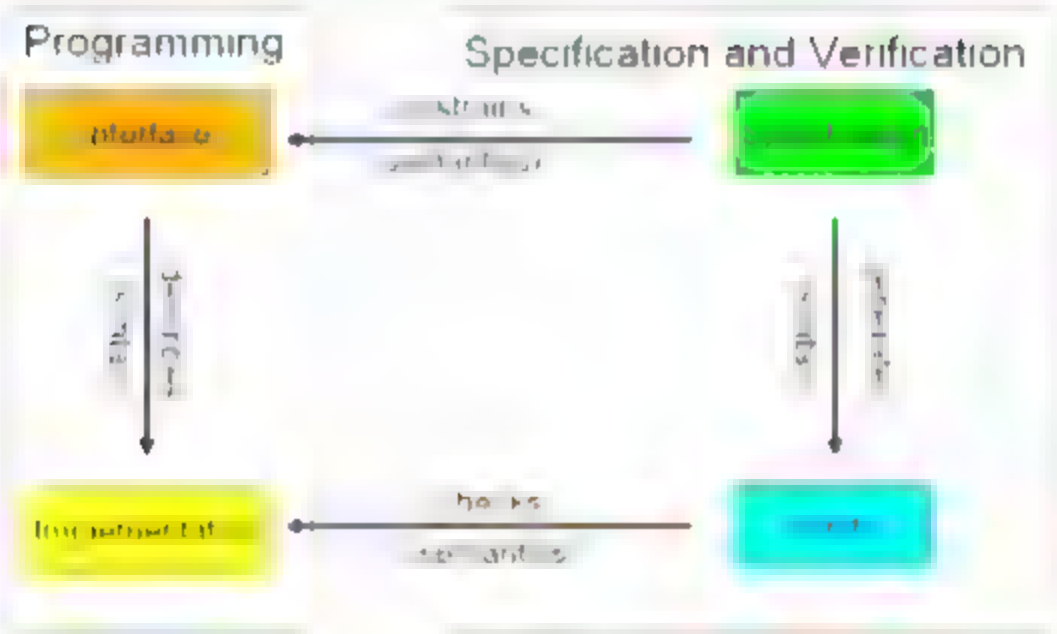
- Hand-written
- Translated from a higher-level specification

AsmL. Microsoft's ASM-Language

A powerful specification language:

- Based on ASMs
- Readable executable testable
- Combines mathematical, object-oriented component-oriented approaches
- Integrated into VS6 (working on VS NET)
- Literate style Word and XML

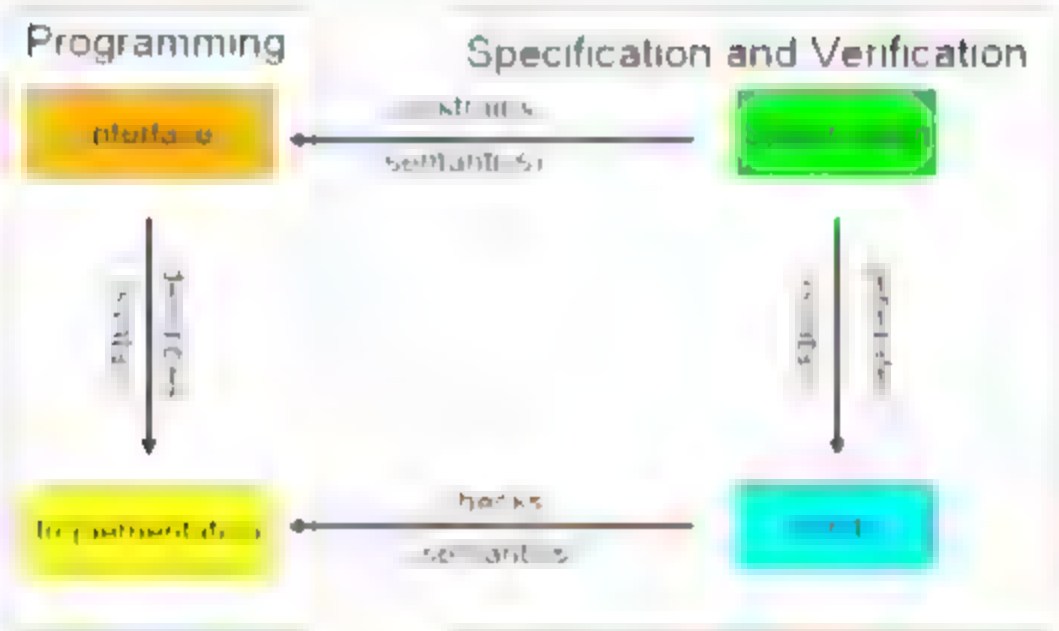
The Big Picture



AsmL Microsoft's ASM-Language

- A powerful specification language:
  - Based on ASMs
  - Readable executable testable
  - Combines mathematical object-oriented component-oriented approaches
  - Integrated into VS6 (working on VS NET)
  - Literate style Word and XML

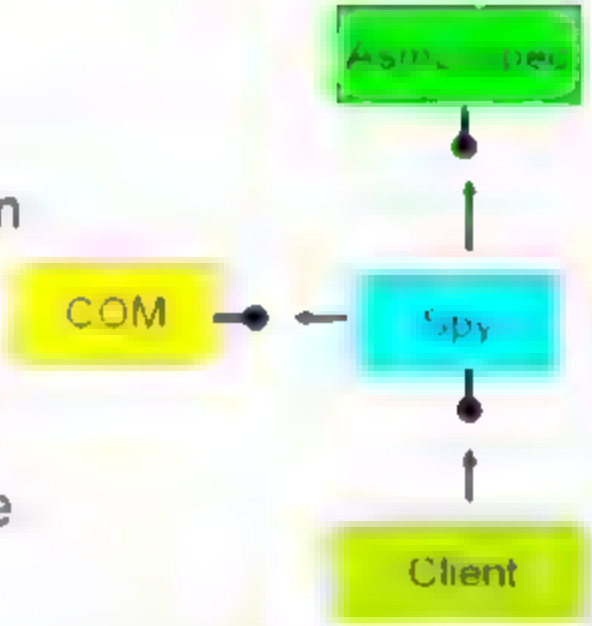
The Big Picture



Previously. .

- Observational equivalence
- No instrumentation required
- Non-determinism
- Callbacks

State equivalence



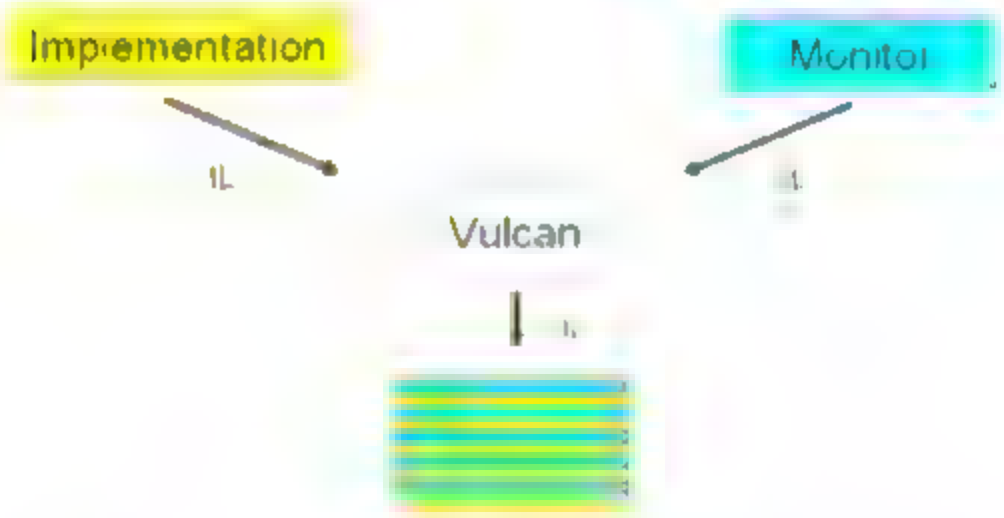
(Trivial) Example

```
interface IMath {
  double sqrt(double x);
}
```

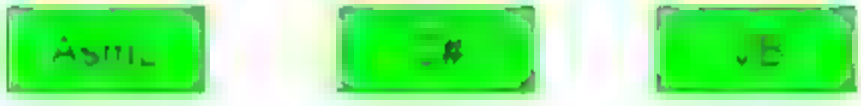
```
class CMath : IMath {
  double sqrt(double x)
    {     }
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    [ x >= 0; ]
  bool sqrt_post(double x)
    { result : result == x }
}
```

Tighter Integration: NET



Monitors



- Hand-written
- Translated from a higher-level specification

Monitor

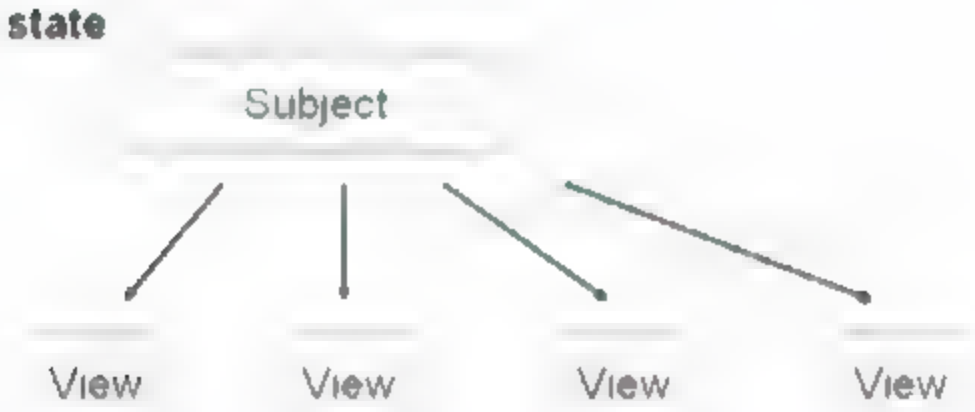
(Trivial) Example

```
interface IMath {
  double sqrt(double x);

}
```

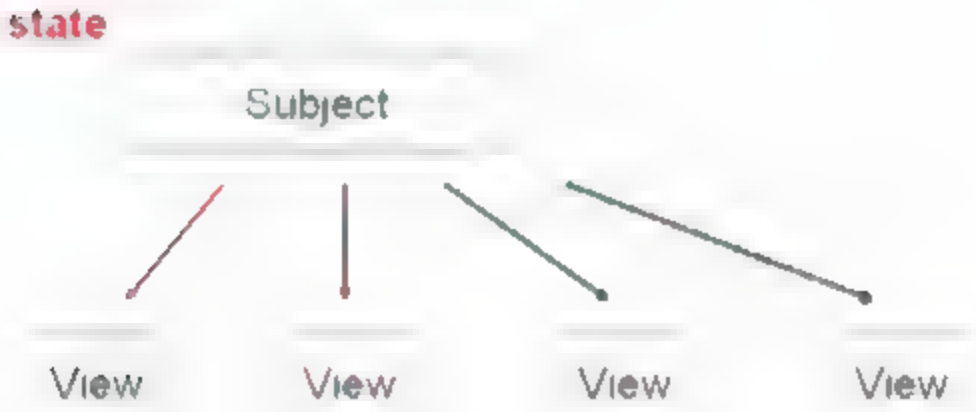
```
class CMath : IMath {
  double sqrt(double x)
    {  }

}
```

```
class IMathContract : CMath {

  bool sqrt_pre(double x)
    { x >= 0; }

  bool sqrt_post(double x)
    { result * result == x }
}
```

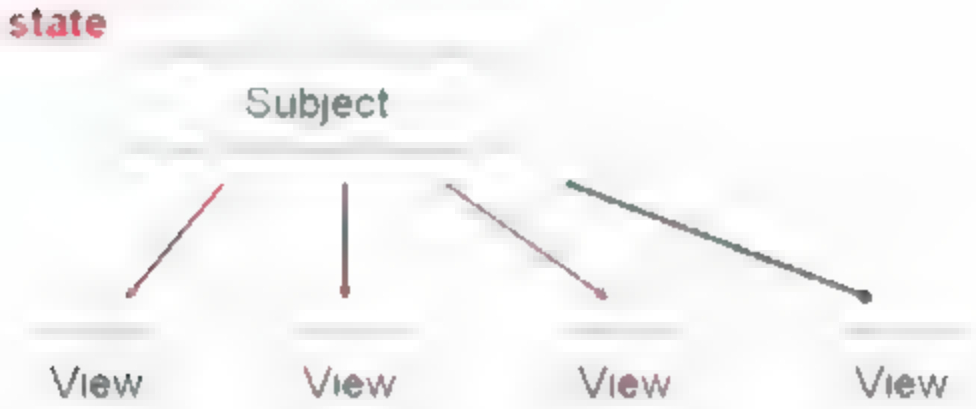
Subject/View



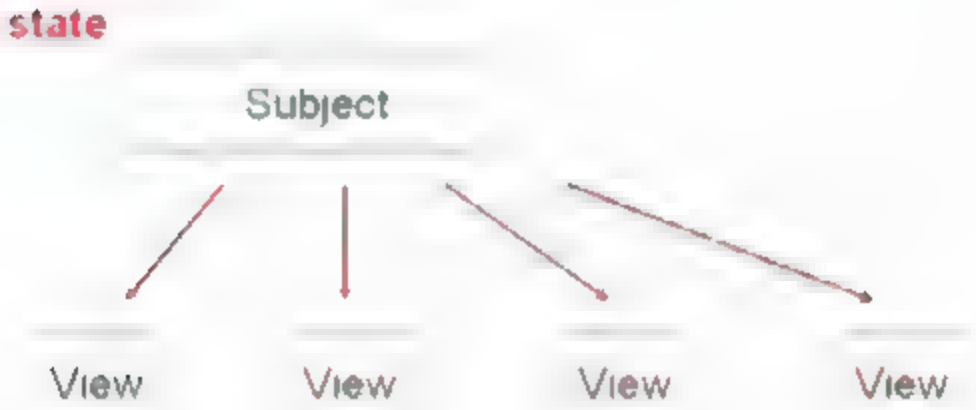
Subject/View



Subject/View



Subject/View



Subject/View in AsmL

```
class Subject[T]
  state as T
  views as Set[IView]

  Get() as T = state

  Set(x as T)
    machine
      state := x
    step
      forall v in views do
        v.Update()
```

Subject/View in AsmL

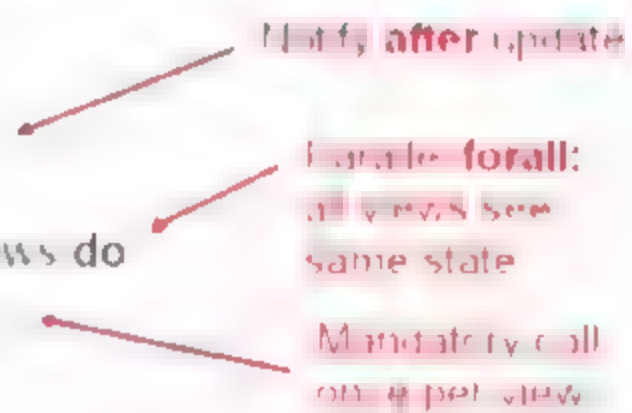
```
class Subject[T]
  state as T
  views as Set[IView]

  Get() as T = state

  Set(x as T)
    machine
      state = x
    step
      forall v in views do
        v.Update()
```

Notify **after** update

forall:
all views see
same state

Subject/View in AsmL

```
class Subject[T]
  state as T
  views as Set[IView]

  Get() as T = state

  Set(x as T)
    machine
      state := x
    step
      forall v in views do
        v.Update()
```



Deriving the Monitor from AsmL

```
Set(x as T) =
  machine  state  = x
  step     forall v in views do v.Update()
```

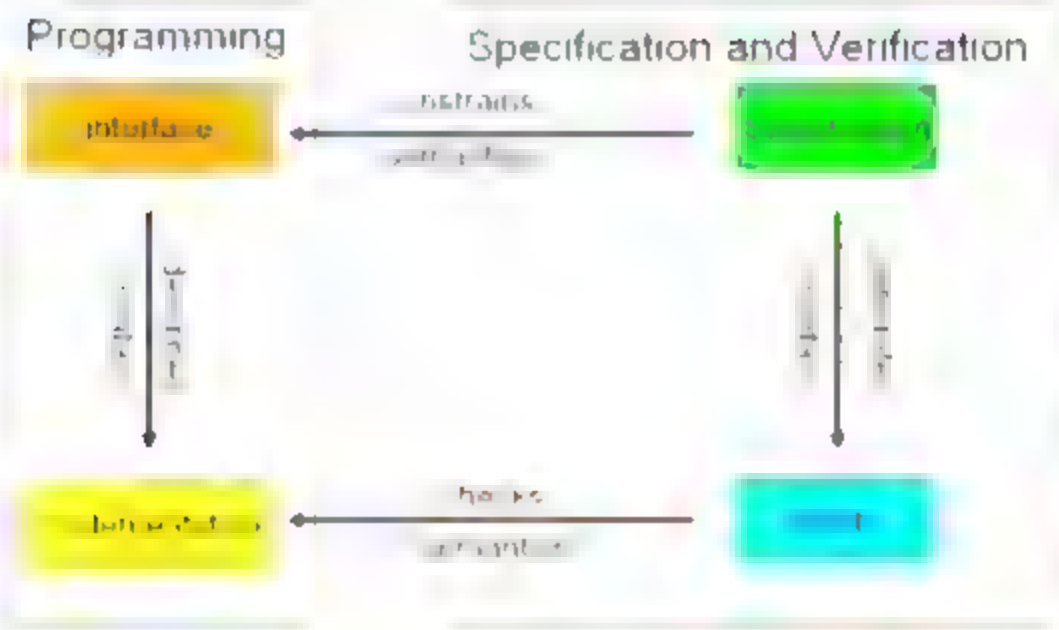
```
Set_pre() { mcalls == { v.Update | v in views }, }
Set_body() { state = x  }
Set_post() { assert(mcalls == {}) }

Update_pre() { assert(state == x) }
Update_post() { mcalls.remove(v.Update) }
```

Summary: Runtime Verification

- Monitor design can be used for arbitrary purposes
- Monitor is generated automatically from AsmL
  - Can be added incrementally
  - Does not require model checking, theorem proving, etc

http://asml

The Big Picture

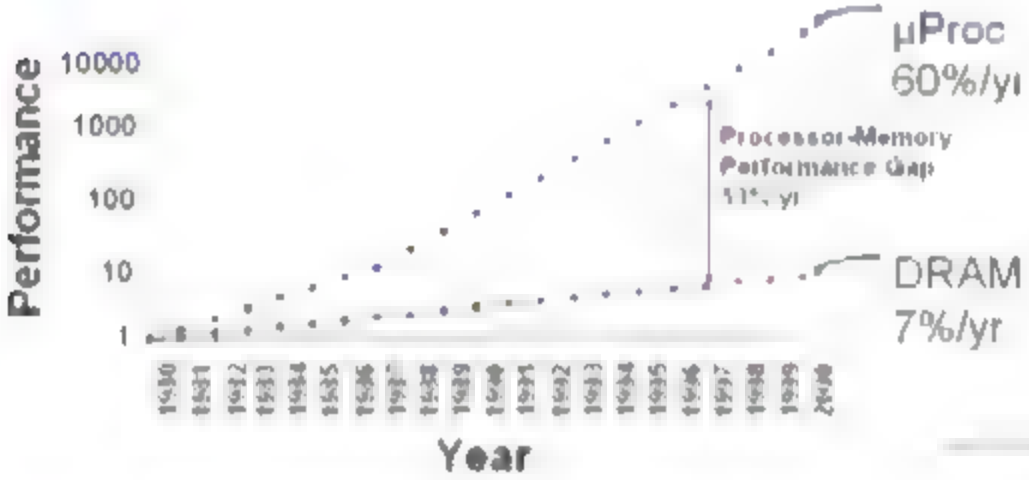


Using Garbage Collection to Improve Program Data Locality

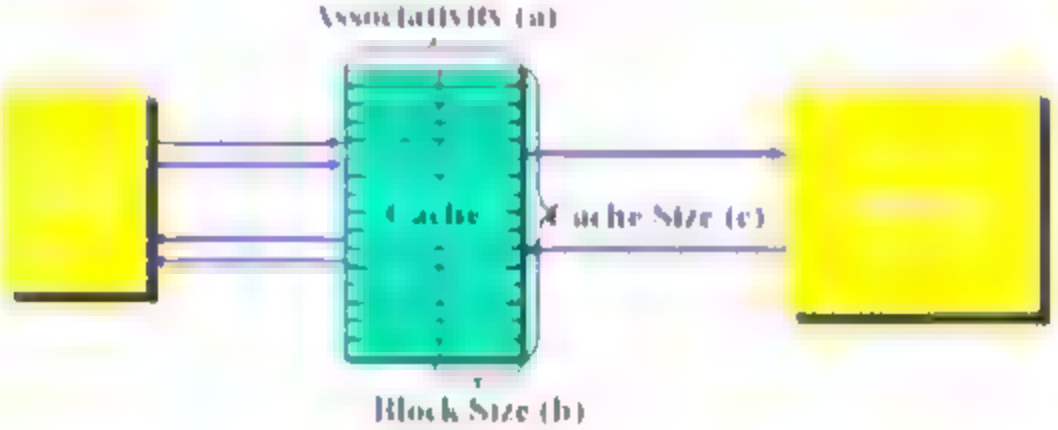
Trishul Chilimbi
PPRC, Performance Monitoring & Analysis
MS Research

http://research.microsoft.com/~trishulc/Daedalus.htm

Processor-Memory Imbalance



Memory Cache: Reduce Latency



Performance Impact

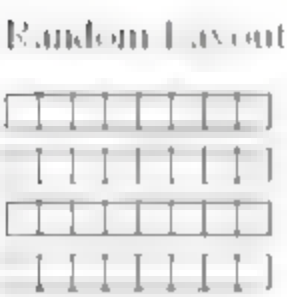
Talk Outline

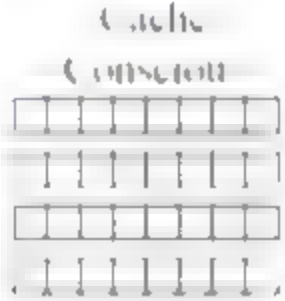
- Motivation
- GC for implementing cache-conscious data layouts
- Low overhead data reference profiling
- Fast and accurate data layout determination

Improving Memory System Performance

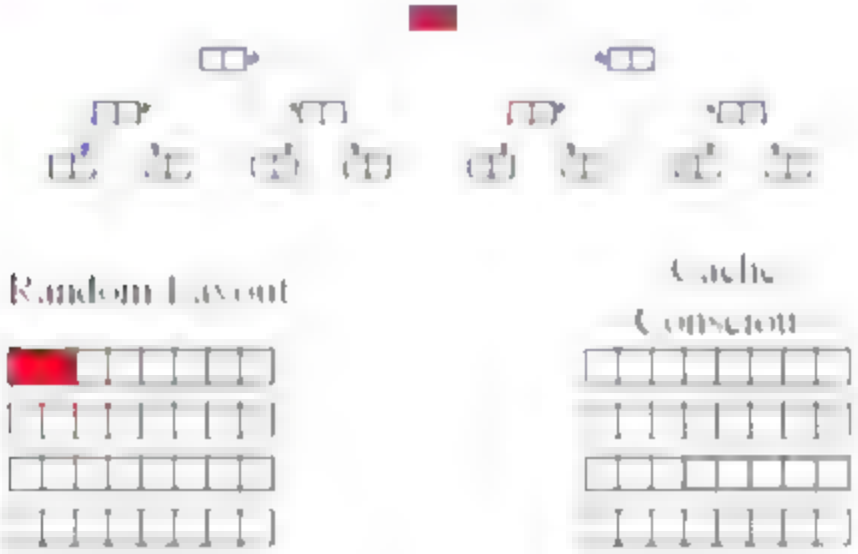
- New Hardware
  - Better caches
- New Software
  - Redesign code and structures
  - Cache-conscious algorithms
- Our approach: Improve locality of existing software automatically
  - Cache-conscious data layouts

Cache-Conscious Data Layout

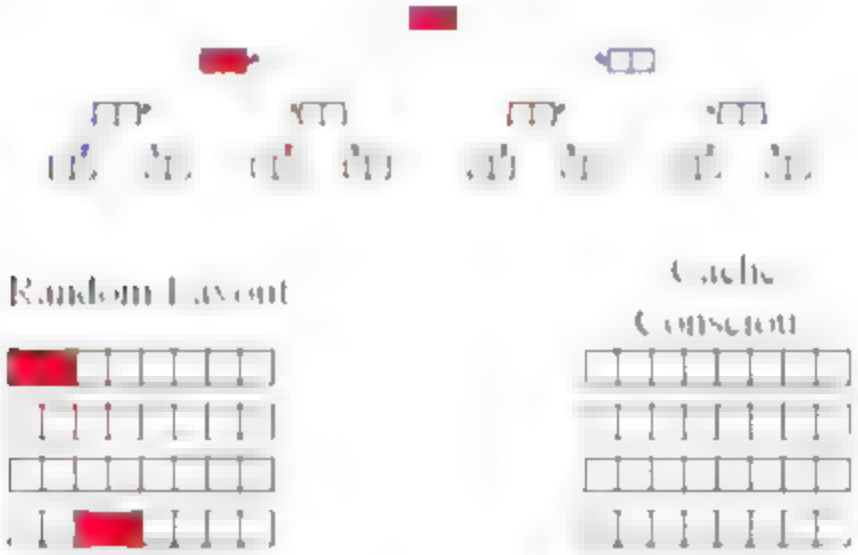




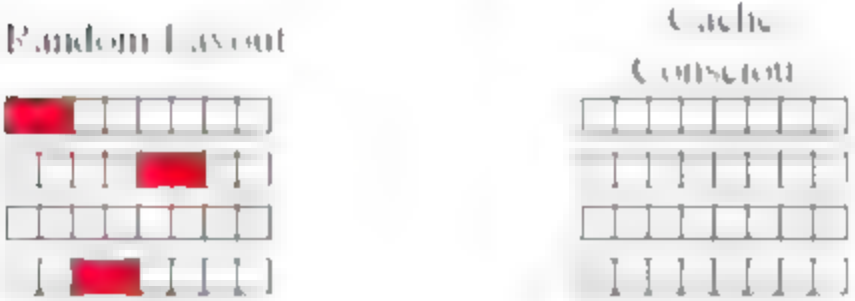
Cache-Conscious Data Layout



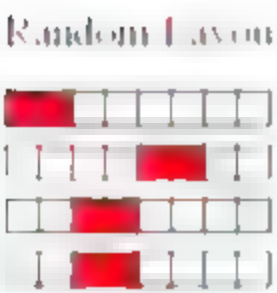
Cache-Conscious Data Layout

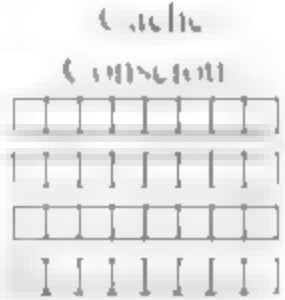


Cache-Conscious Data Layout

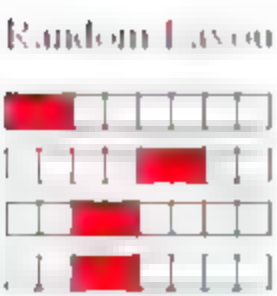


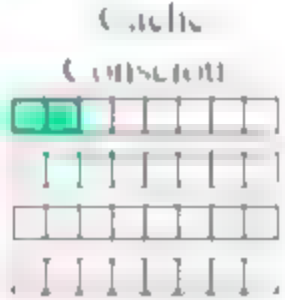
Cache-Conscious Data Layout



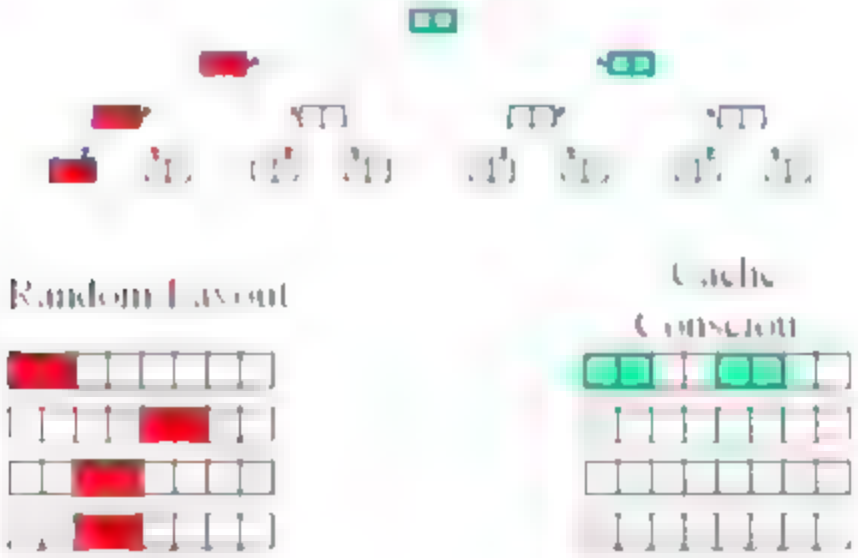


Cache-Conscious Data Layout

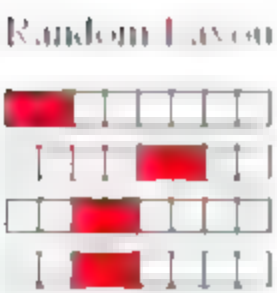


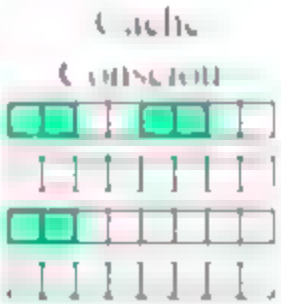


Cache-Conscious Data Layout

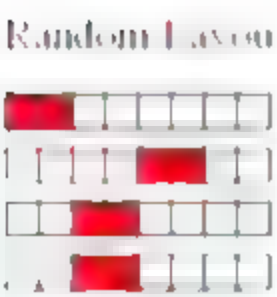


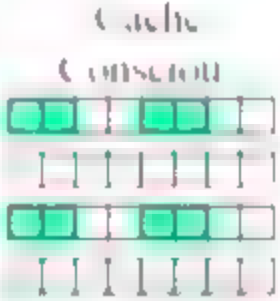
Cache-Conscious Data Layout





Cache-Conscious Data Layout

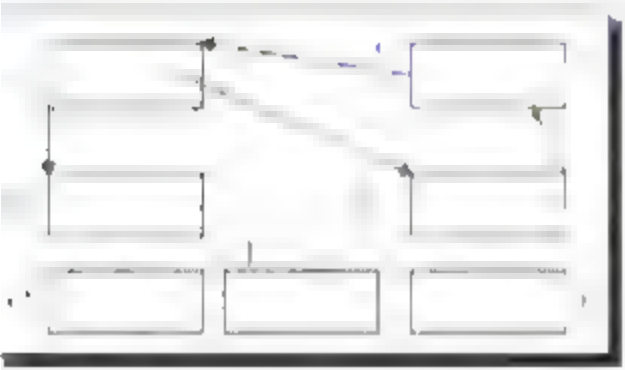


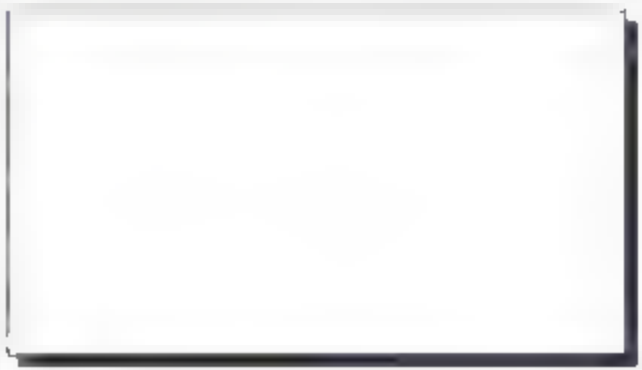


Talk Outline

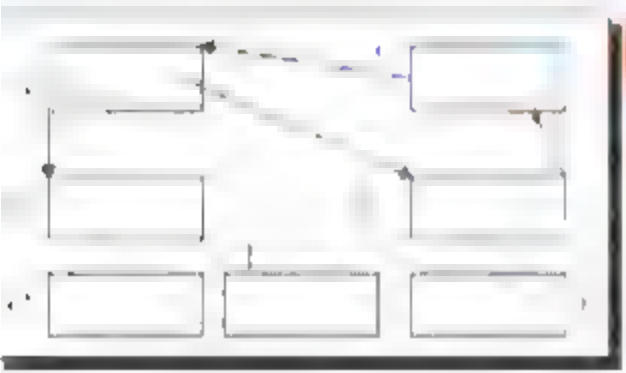
-
- GC for implementing cache-conscious data layouts
  - Structure reorganization
  - Hot/cold splitting
  - Limitations
- Low-overhead data reference profiling
- Fast and accurate data layout determination

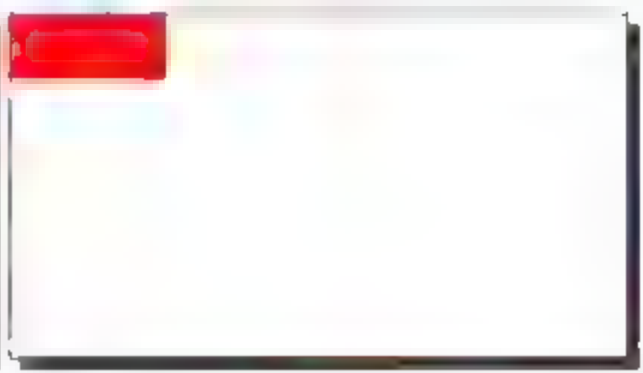
Copying Garbage Collection



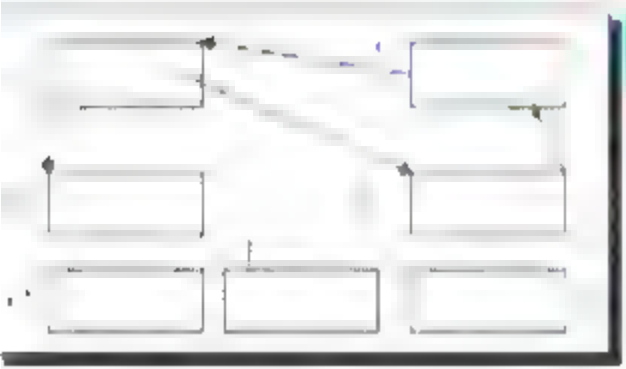


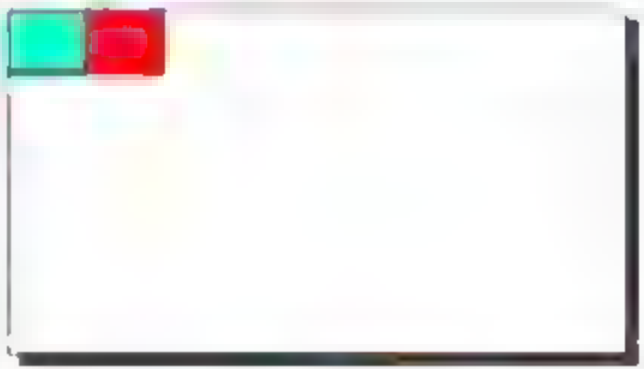
Copying Garbage Collection





Copying Garbage Collection

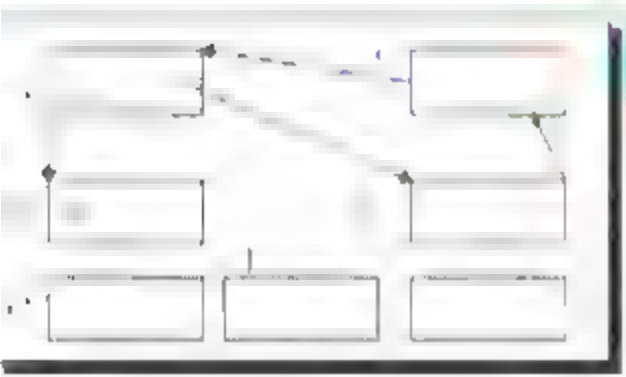


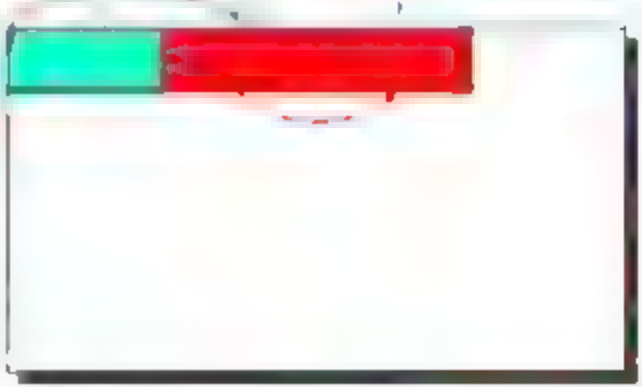


Copying Garbage Collection

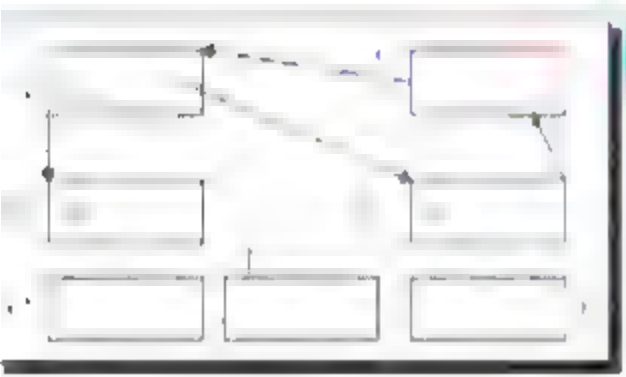
Copying Garbage Collection

Copying Garbage Collection

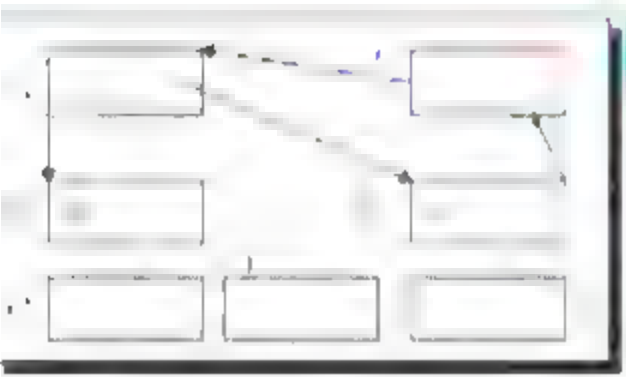


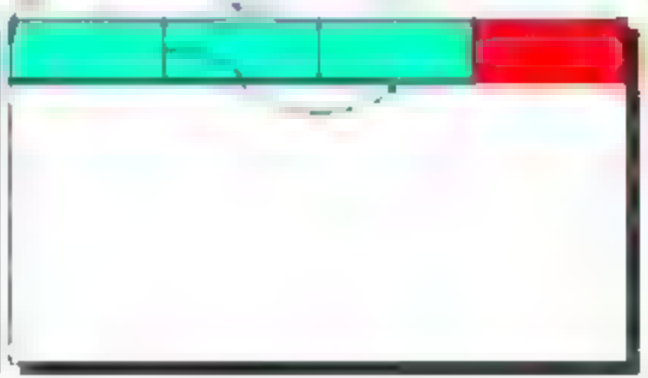


Copying Garbage Collection



Copying Garbage Collection





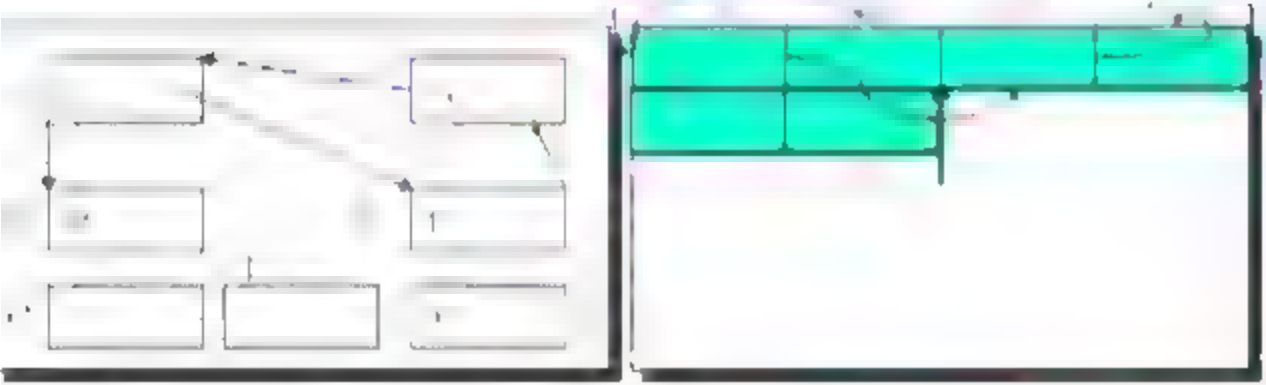
Copying Garbage Collection

Copying Garbage Collection

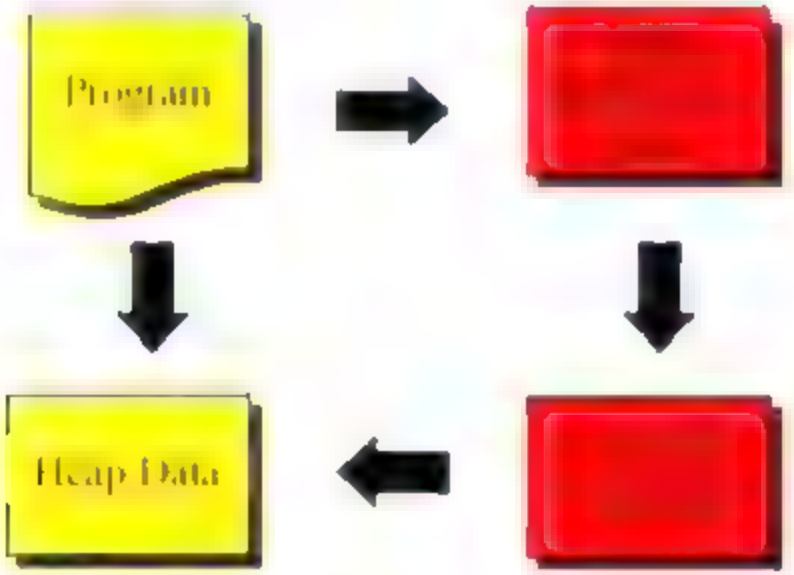
Copying Garbage Collection

Copying Garbage Collection

Copying Garbage Collection



Using Copying GC for Cache Conscious Reorganization

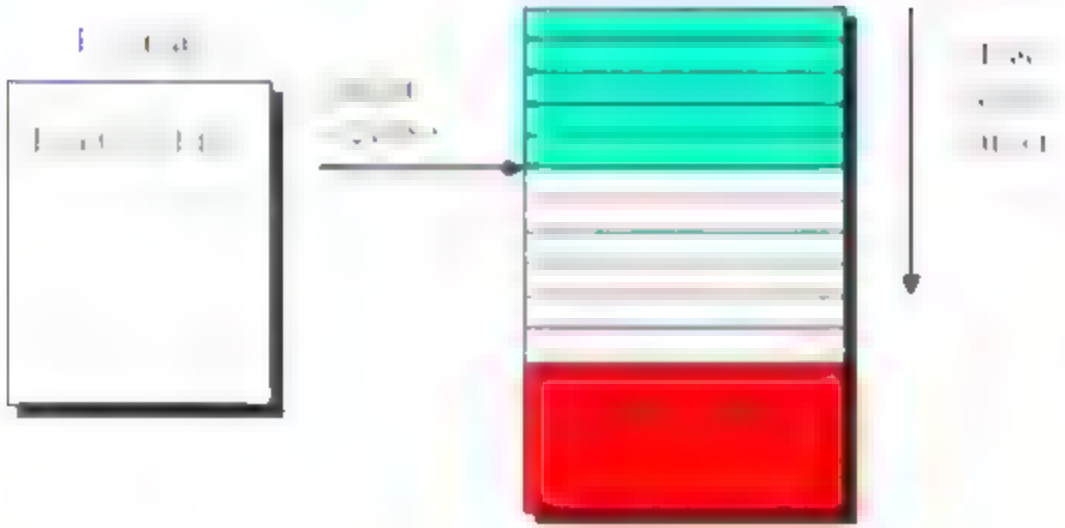


Object Access Buffer

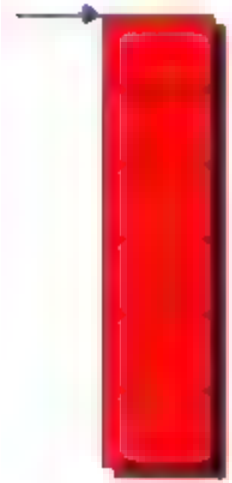
Object Access Buffer

Object Access Buffer

Object Access Buffer



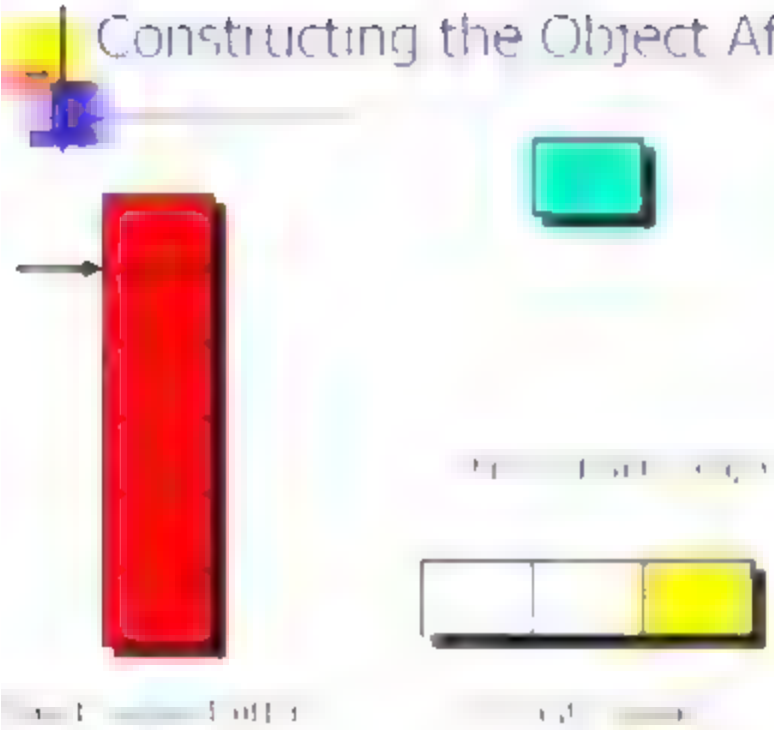
Constructing the Object Affinity Graph



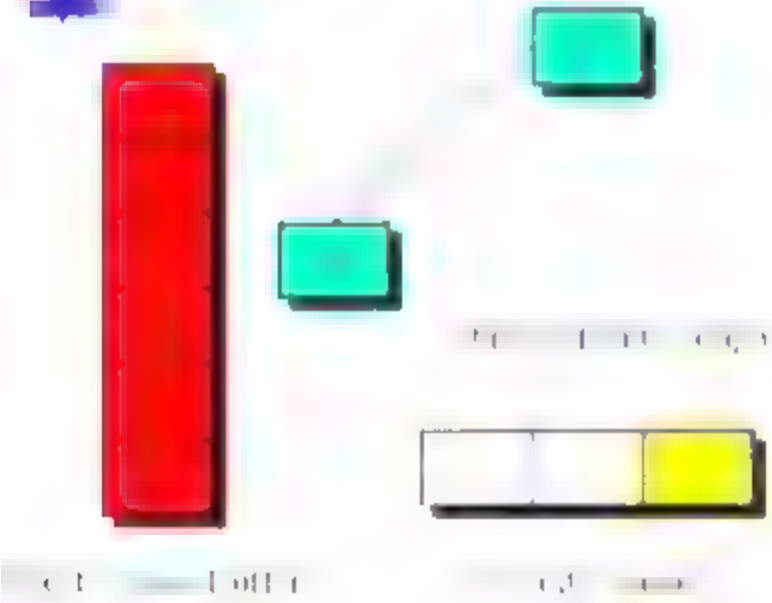
Next Access Pattern

Result

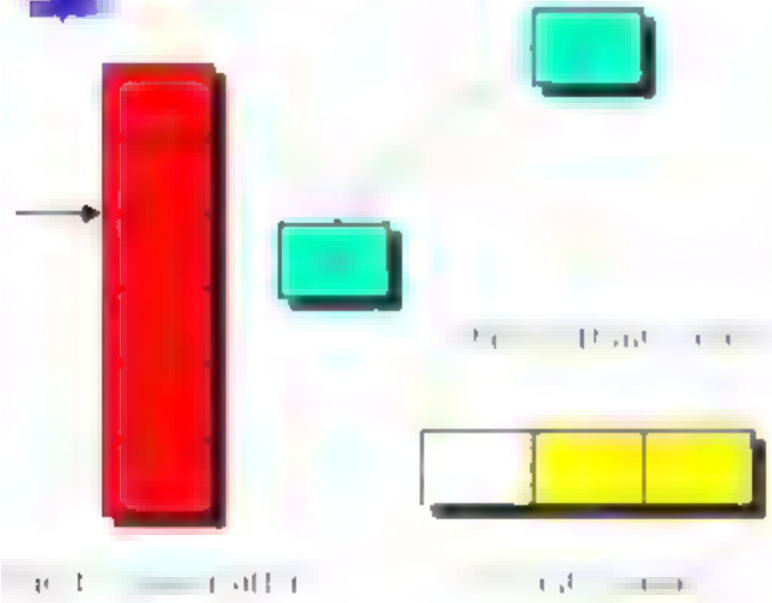
Constructing the Object Affinity Graph



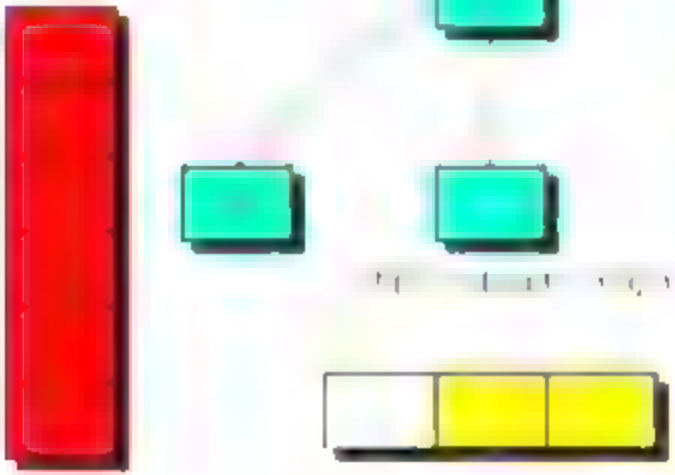
Constructing the Object Affinity Graph



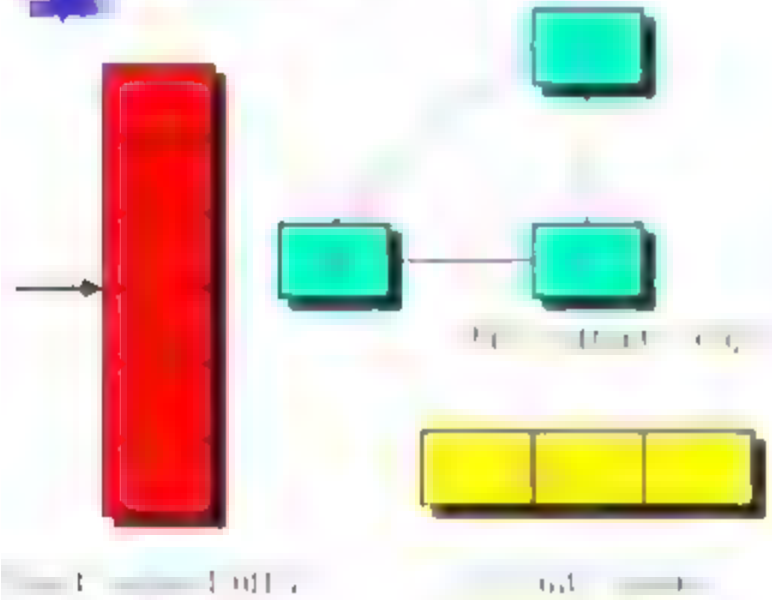
Constructing the Object Affinity Graph



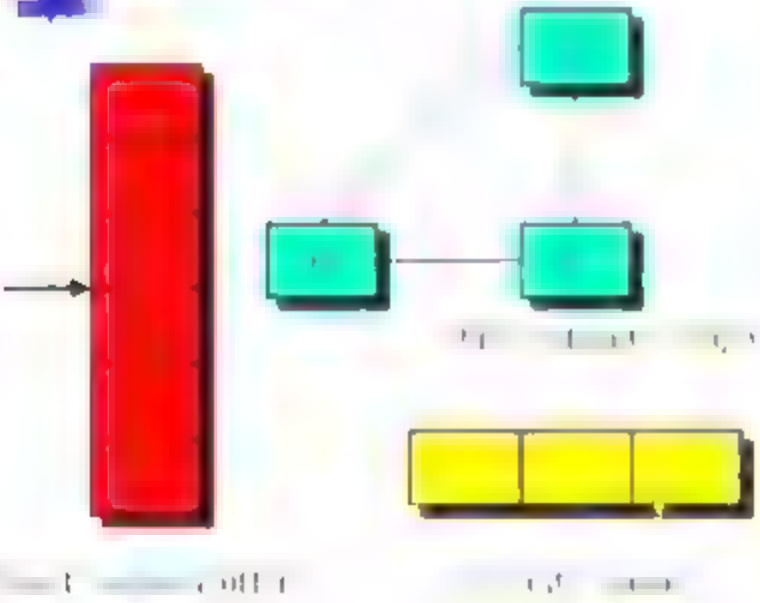
Constructing the Object Affinity Graph



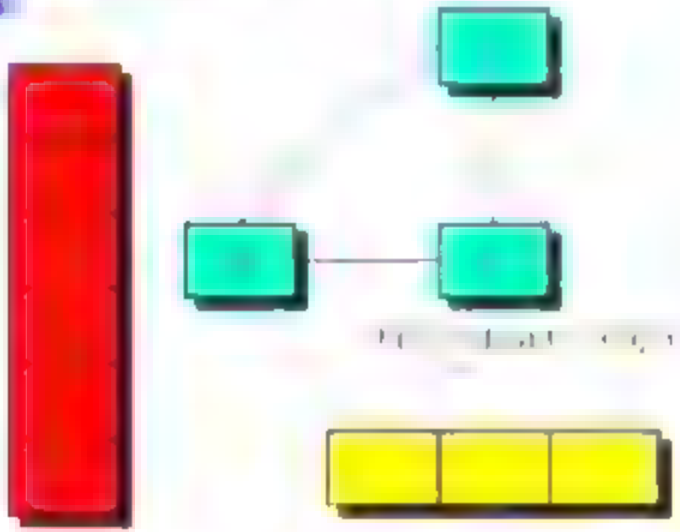
Constructing the Object Affinity Graph



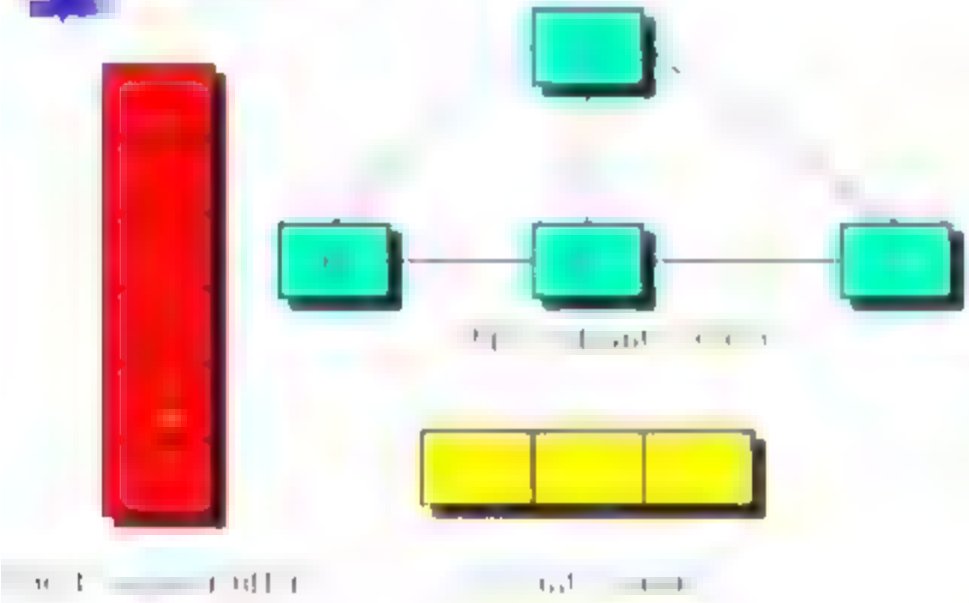
Constructing the Object Affinity Graph



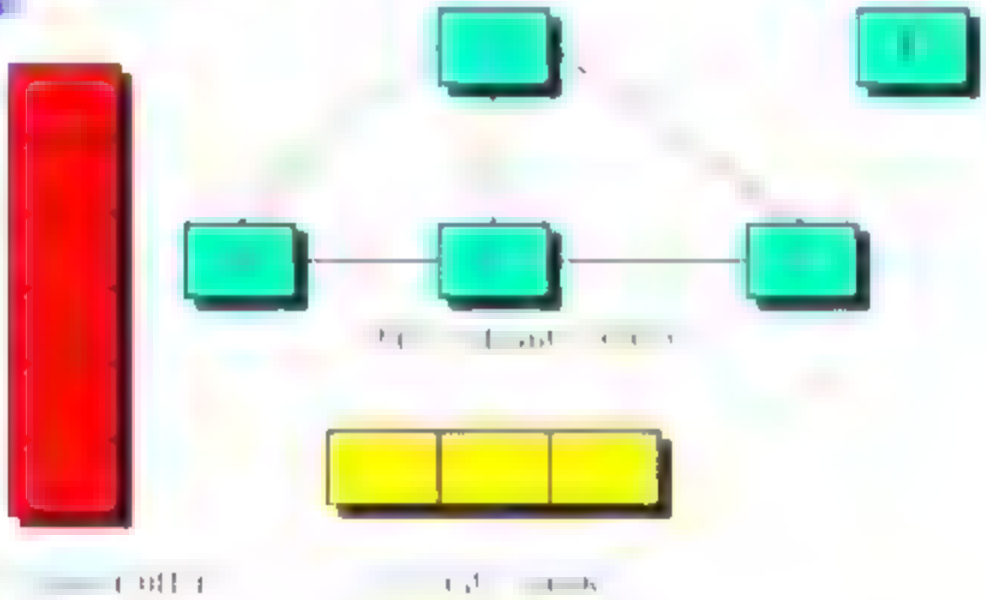
Constructing the Object Affinity Graph



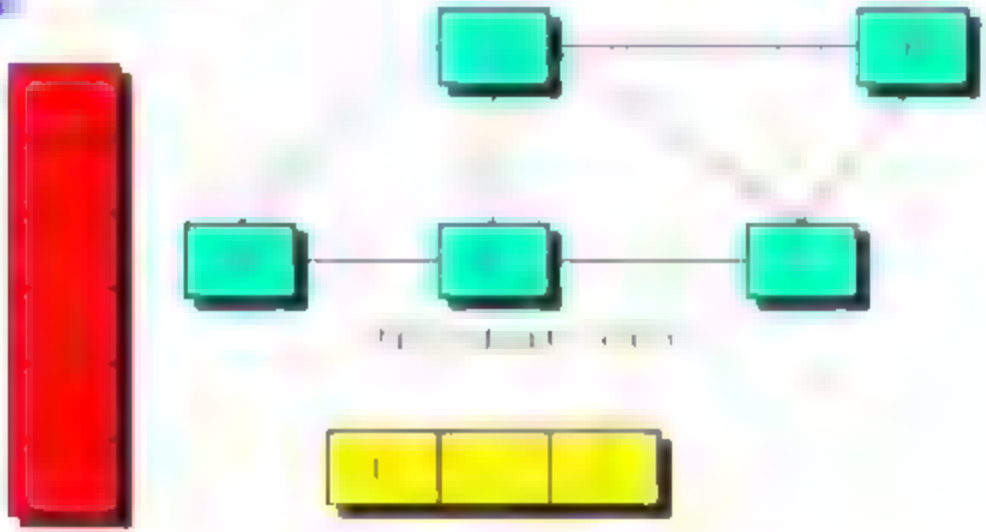
Constructing the Object Affinity Graph



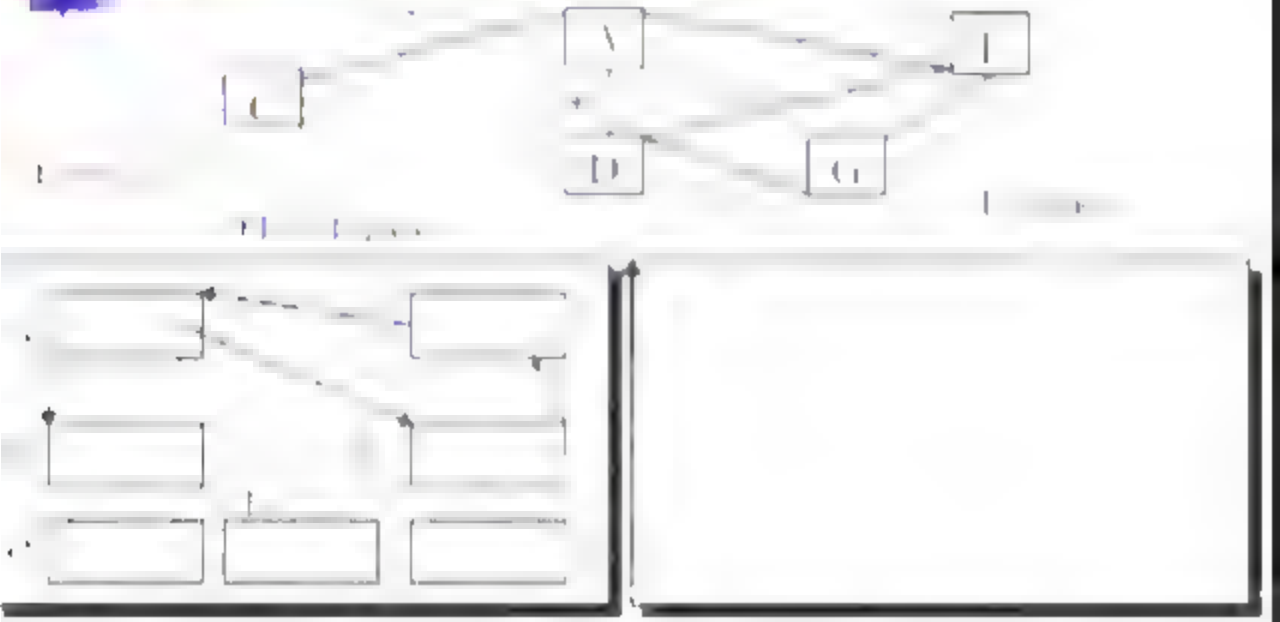
Constructing the Object Affinity Graph



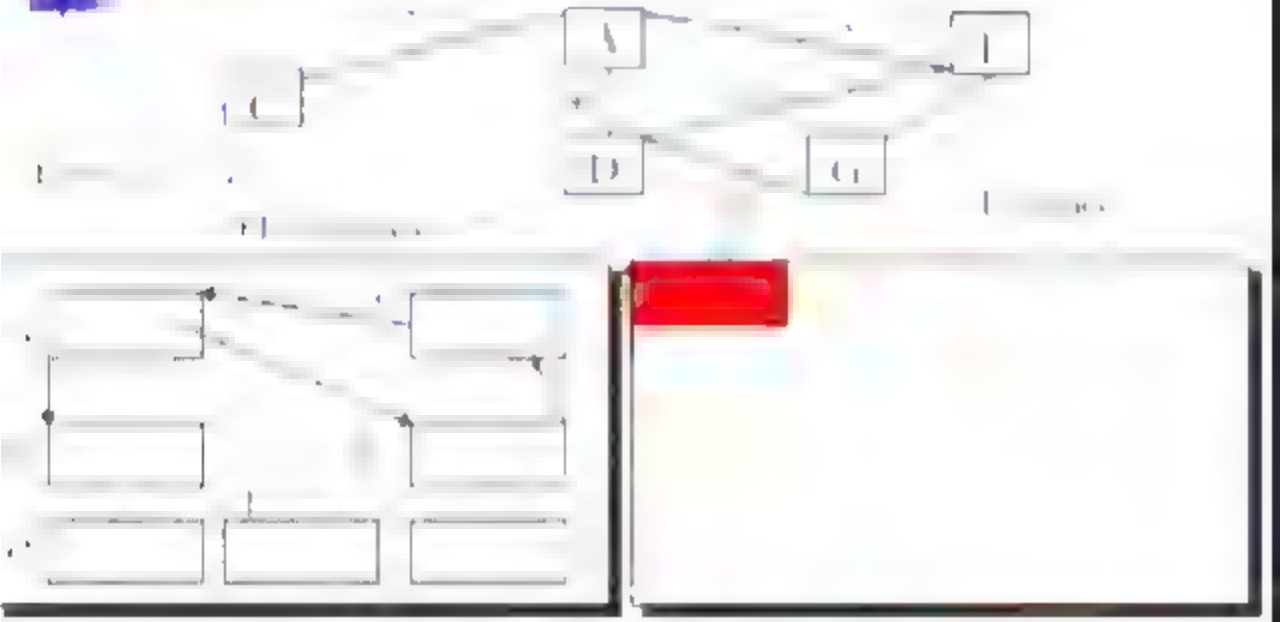
Constructing the Object Affinity Graph



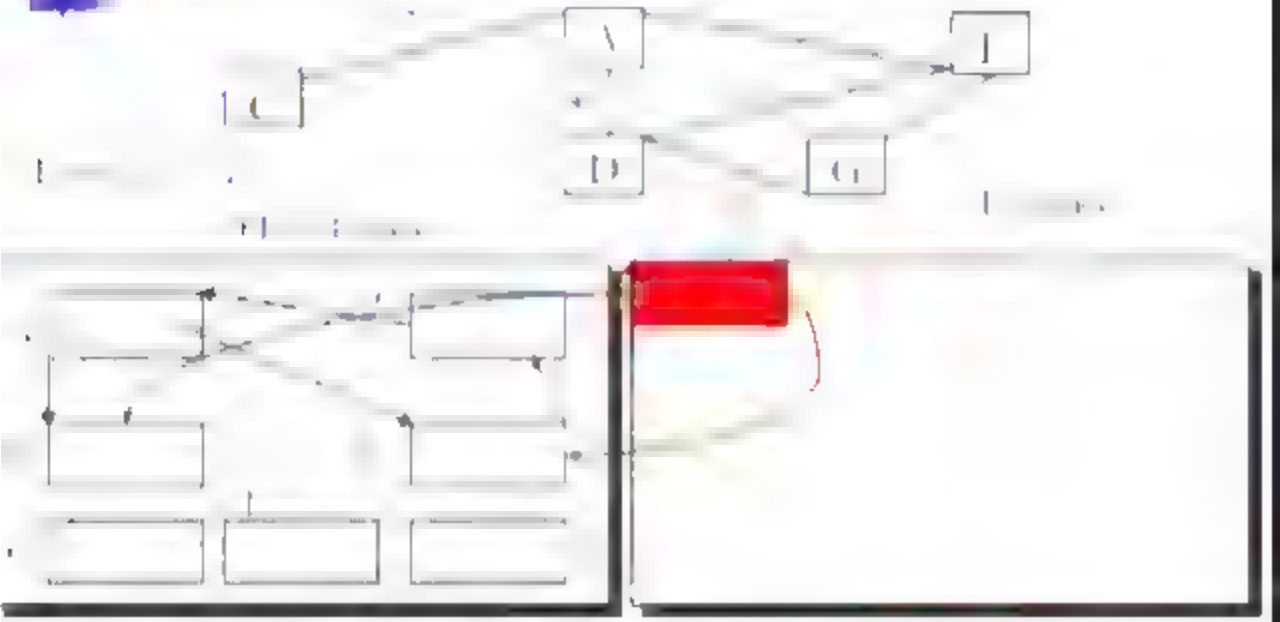
Cache-Conscious Copying



Cache-Conscious Copying



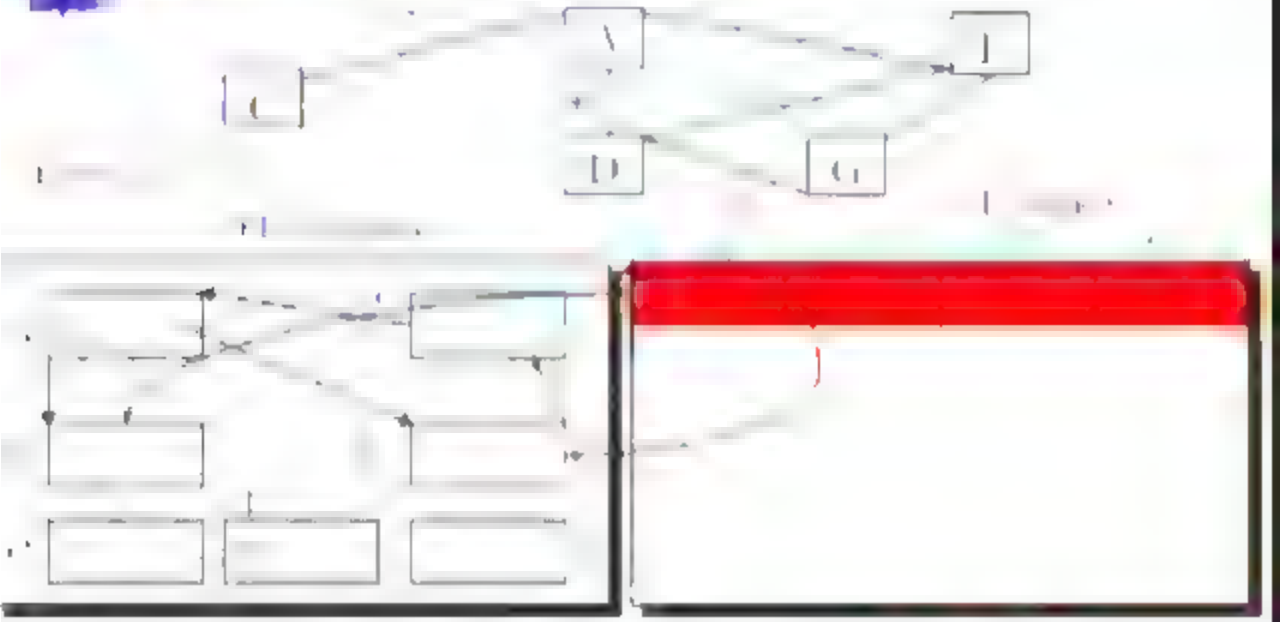
Cache-Conscious Copying



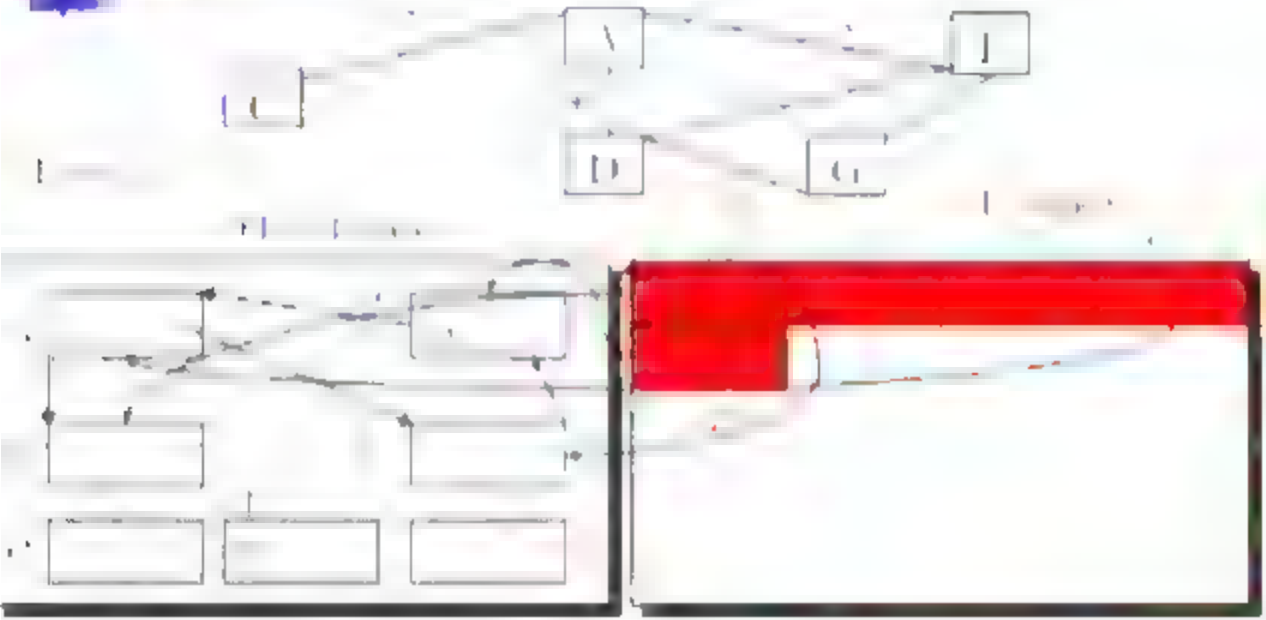
Cache-Conscious Copying

Cache-Conscious Copying

Cache-Conscious Copying

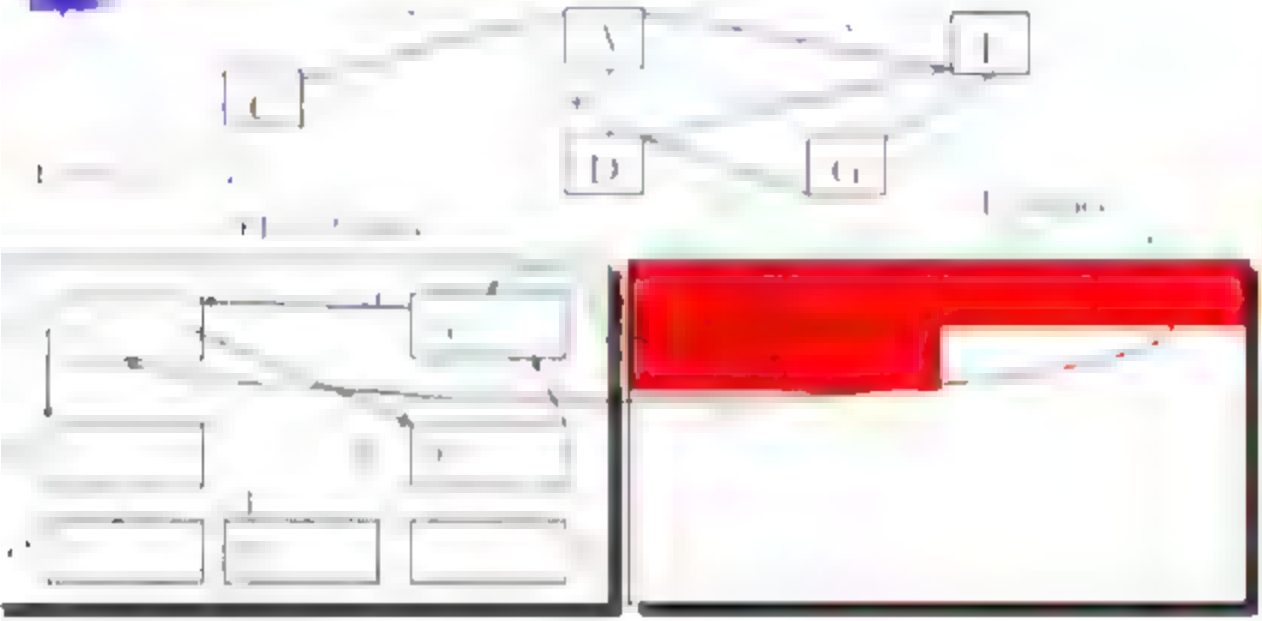


Cache-Conscious Copying



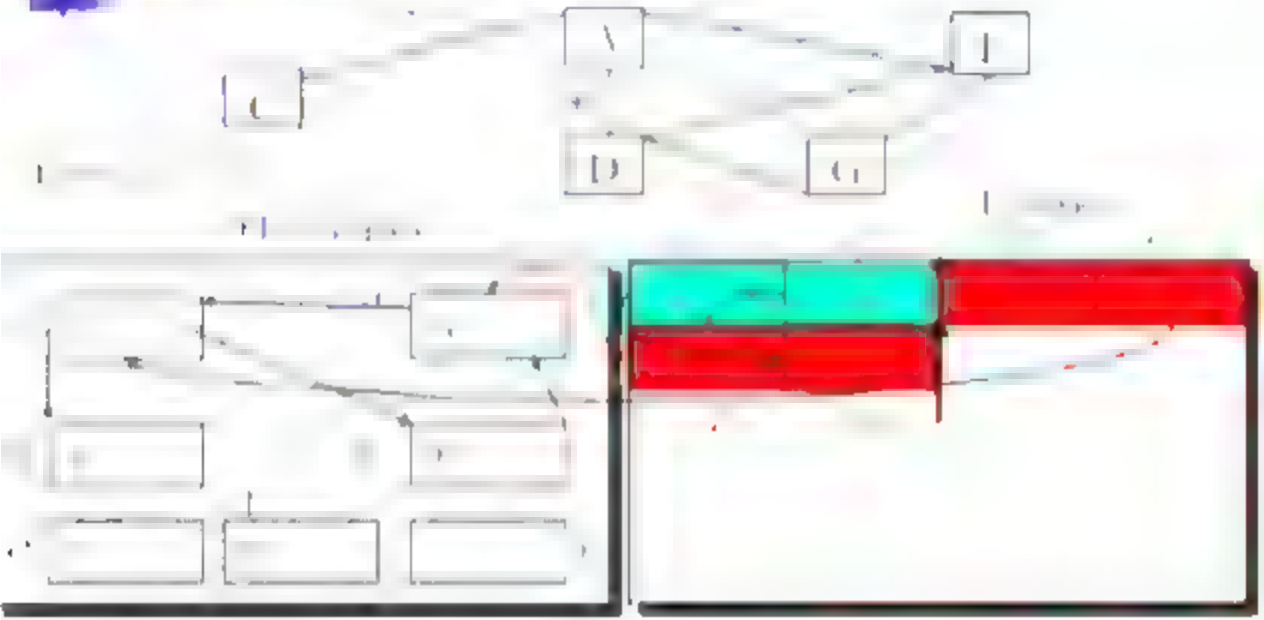
Cache-Conscious Copying

Cache-Conscious Copying

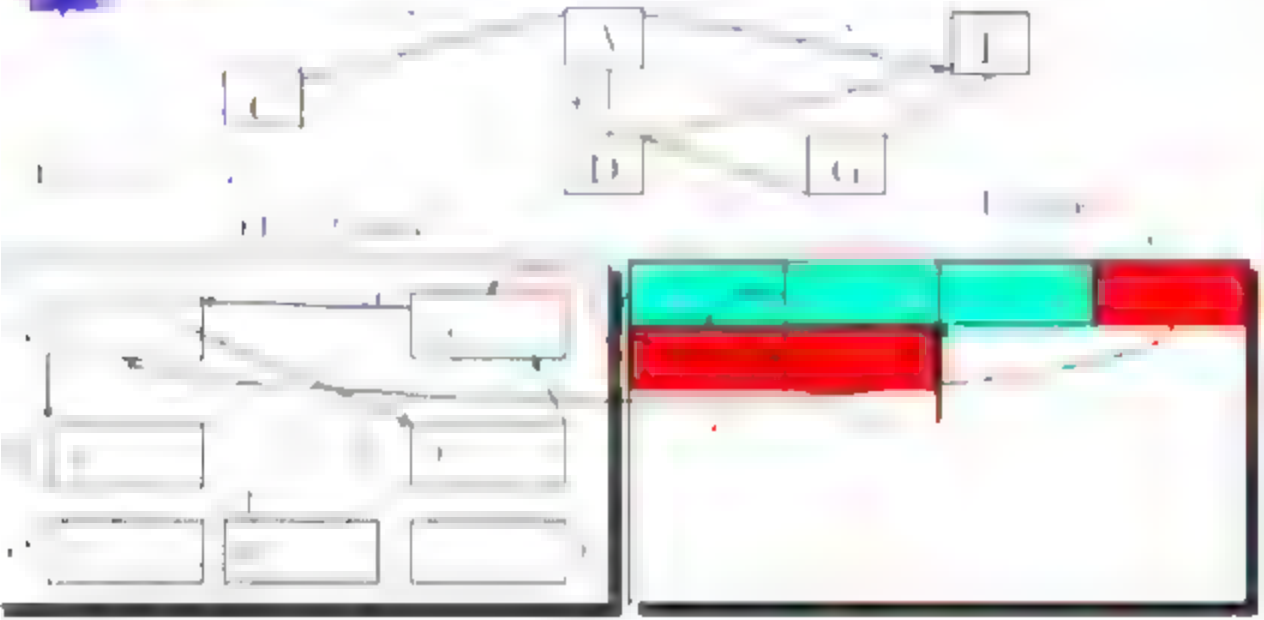


Cache-Conscious Copying

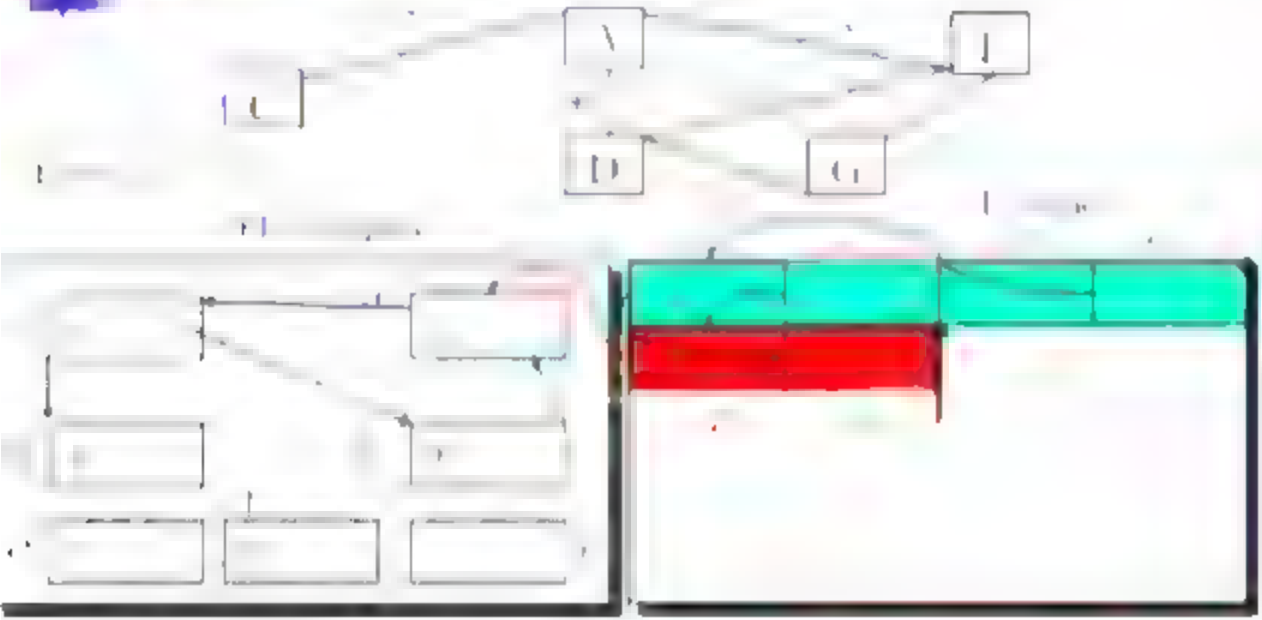
Cache-Conscious Copying



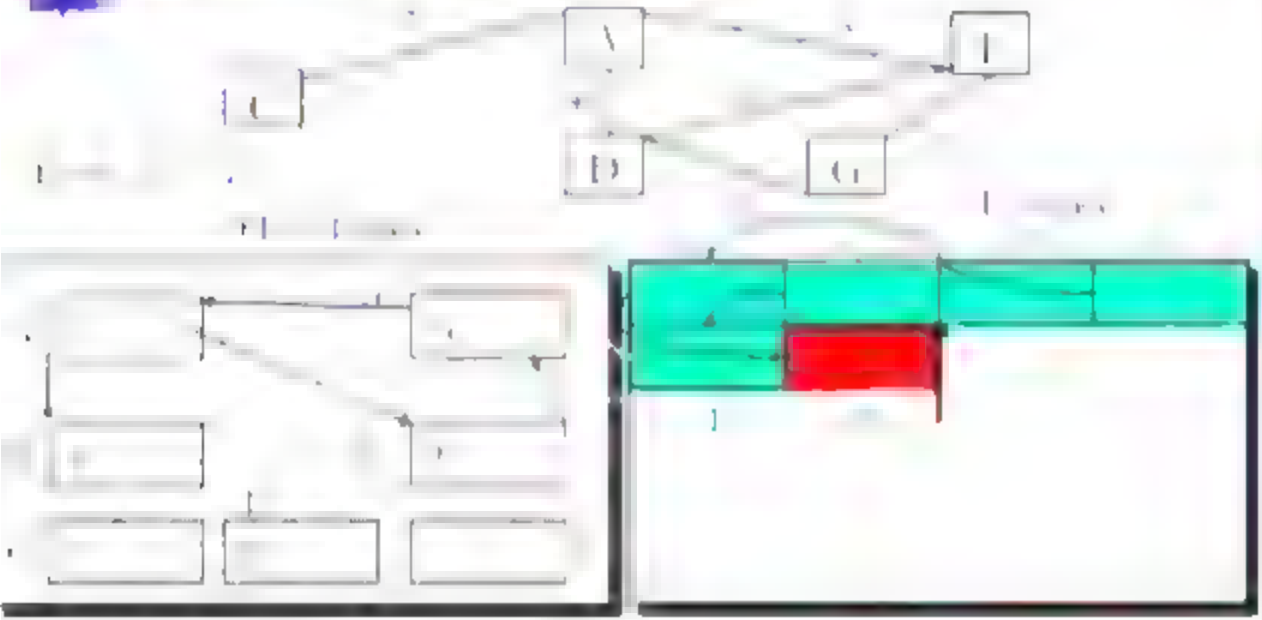
Cache-Conscious Copying



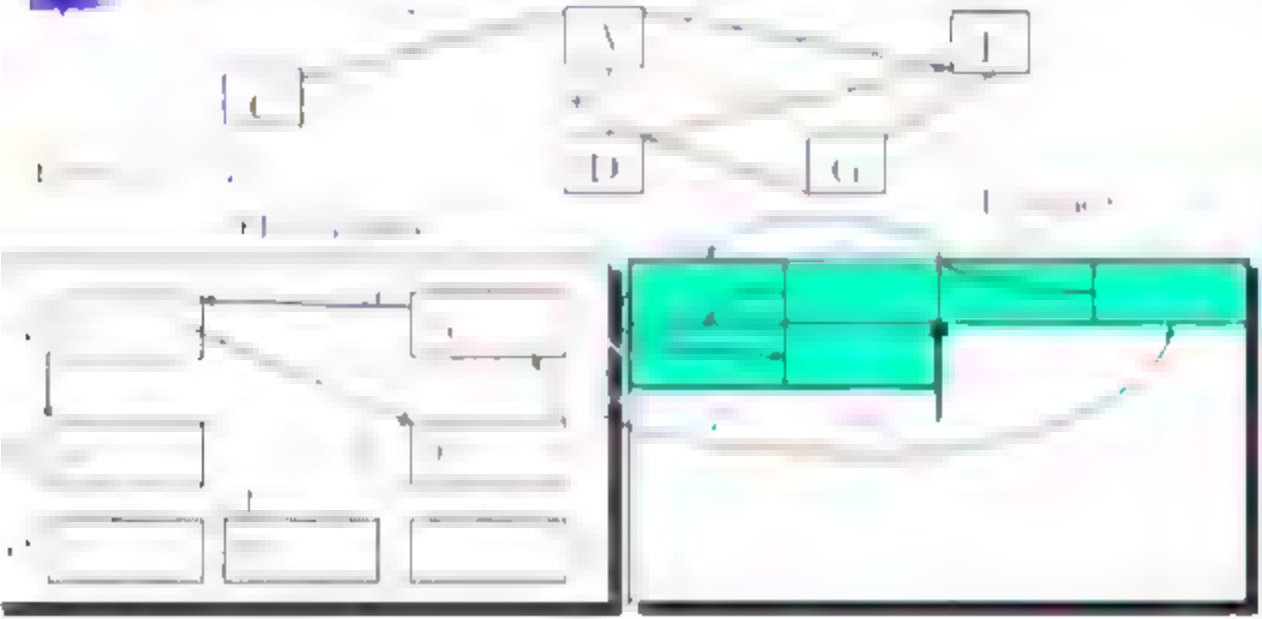
Cache-Conscious Copying



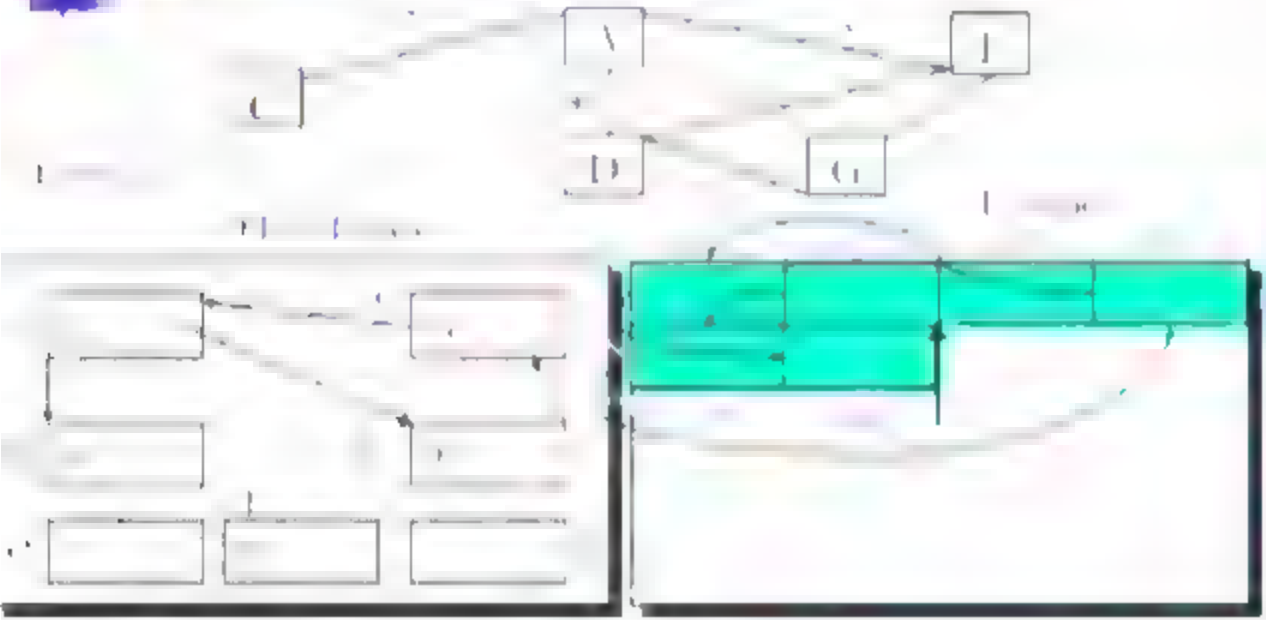
Cache-Conscious Copying



Cache-Conscious Copying

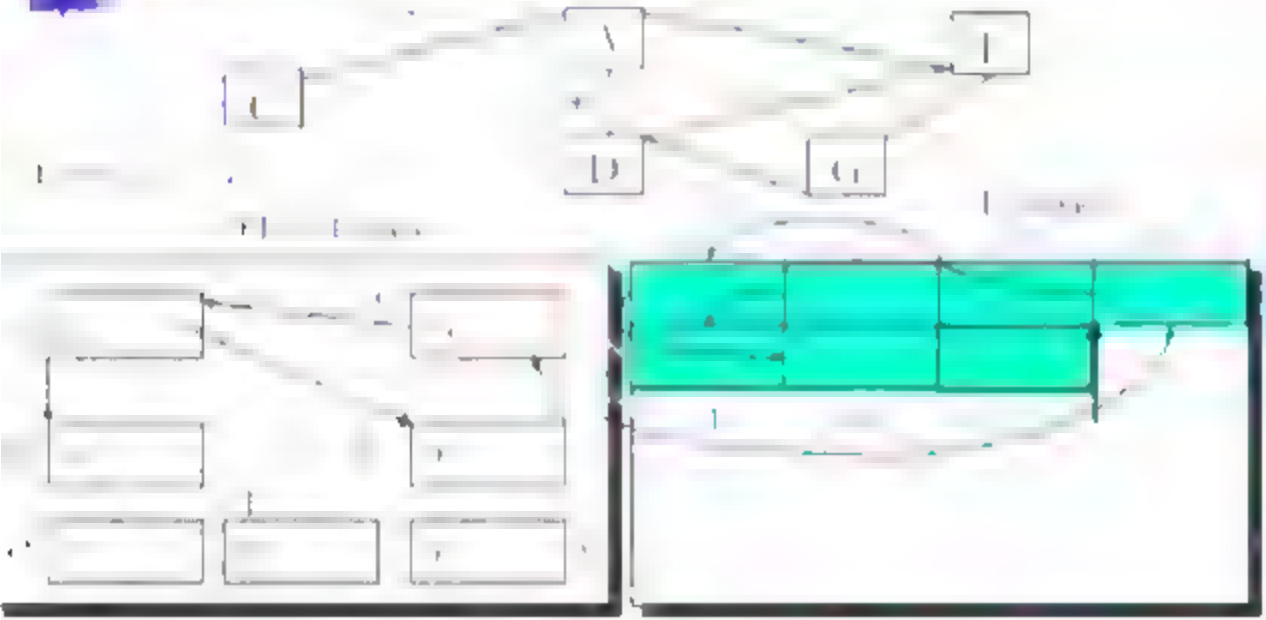


Cache-Conscious Copying

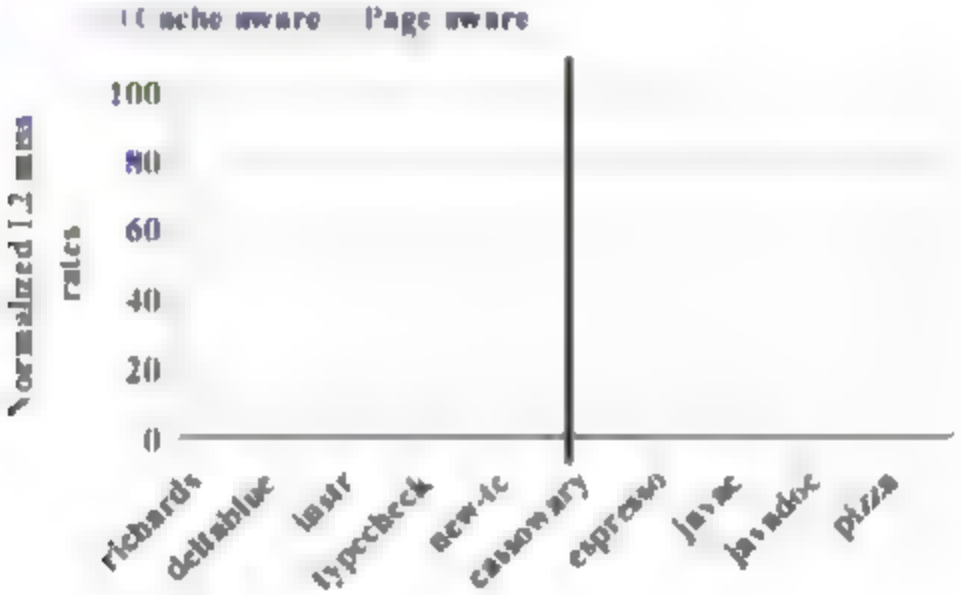


Cache-Conscious Copying

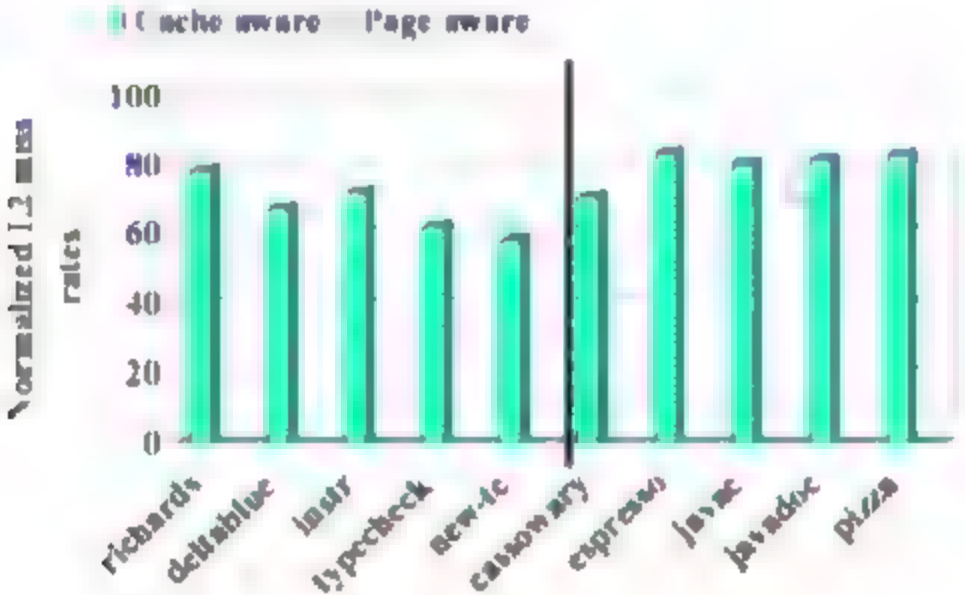
Cache-Conscious Copying



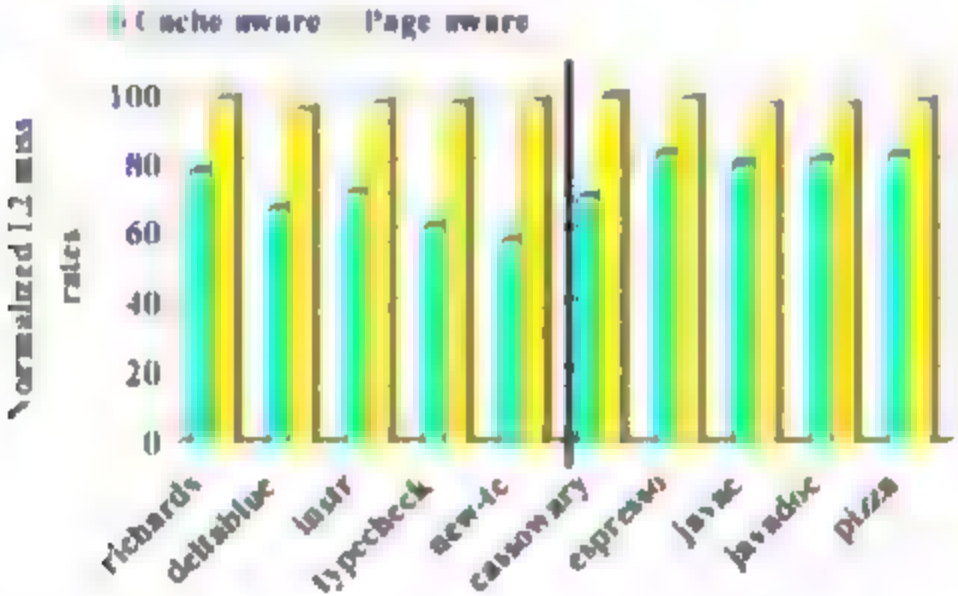
Cache-Conscious Reorganization



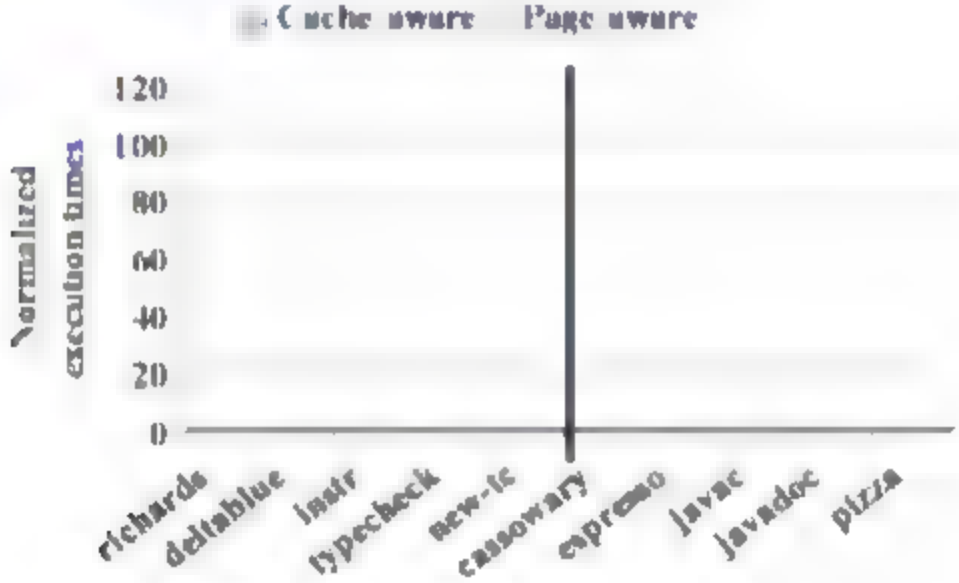
Cache-Conscious Reorganization



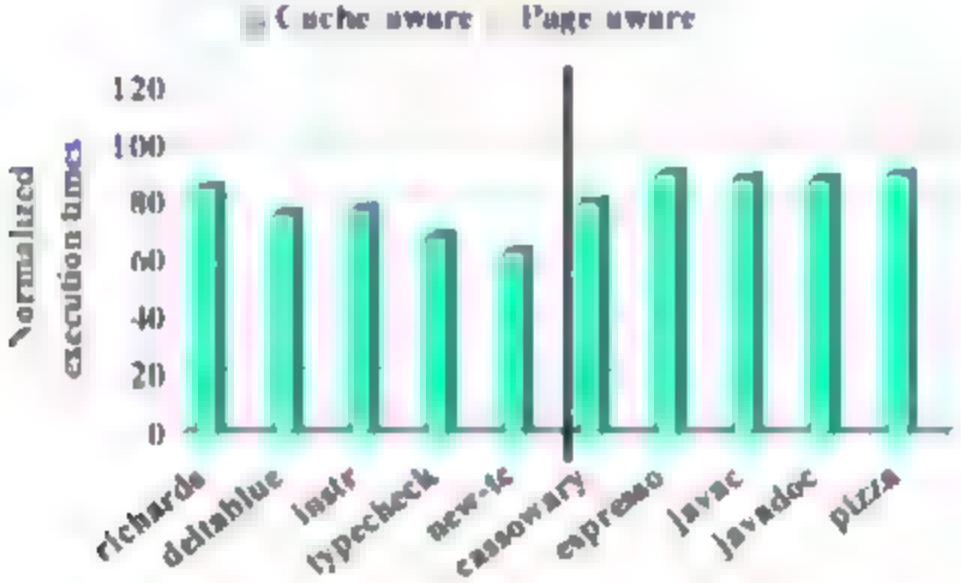
Cache-Conscious Reorganization



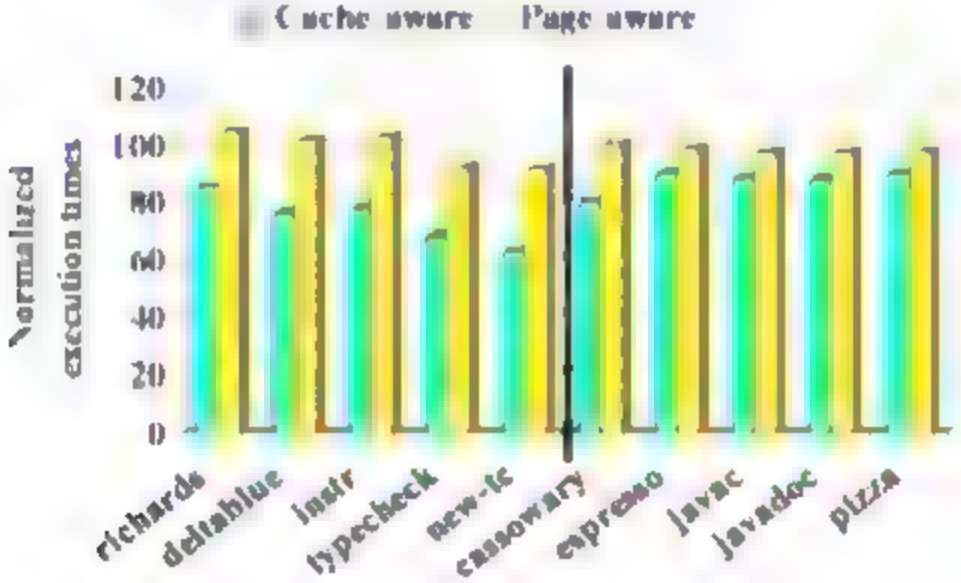
Cache-Conscious Reorganization



Cache-Conscious Reorganization

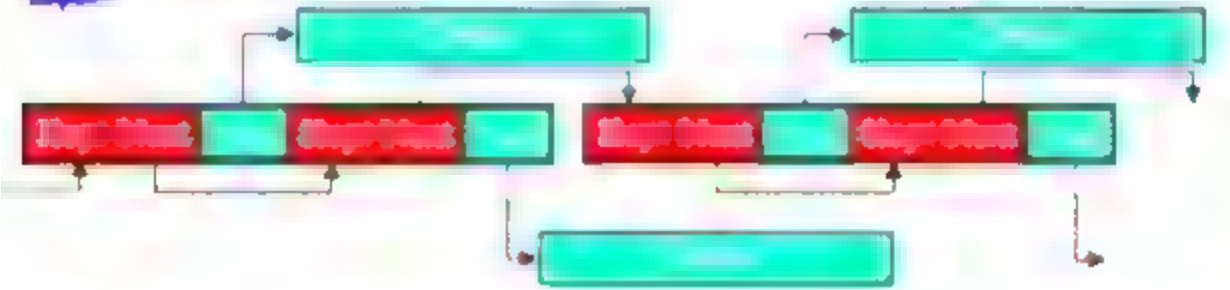


Cache-Conscious Reorganization

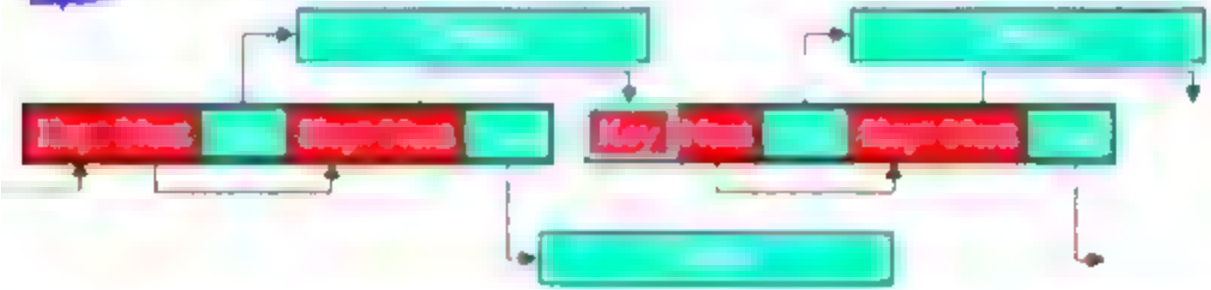


Hot/Cold Structure Splitting

Hot/Cold Structure Splitting



Hot/Cold Structure Splitting

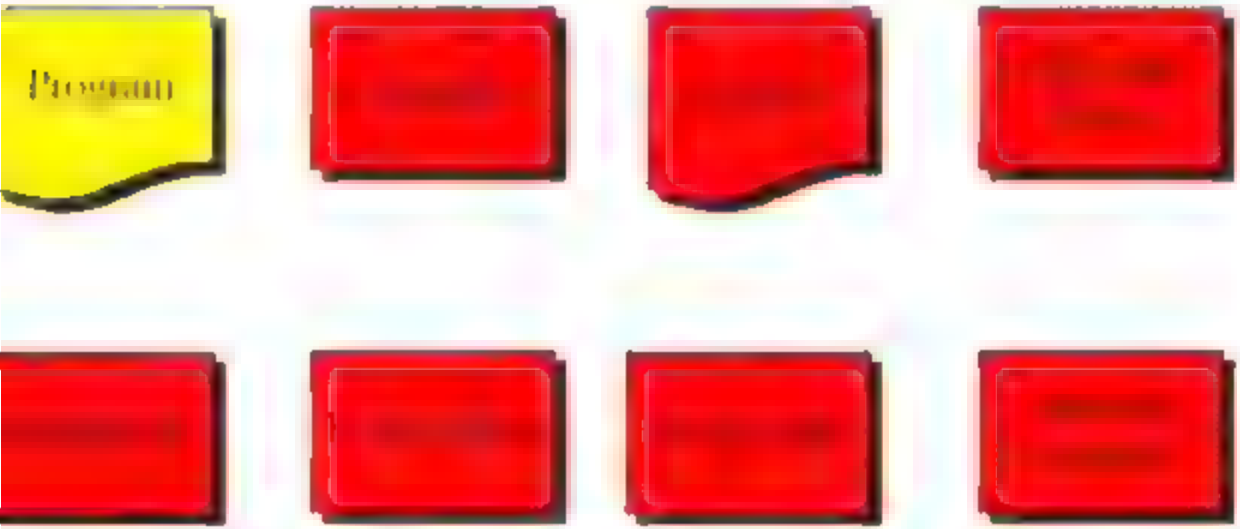


Hot/Cold Structure Splitting

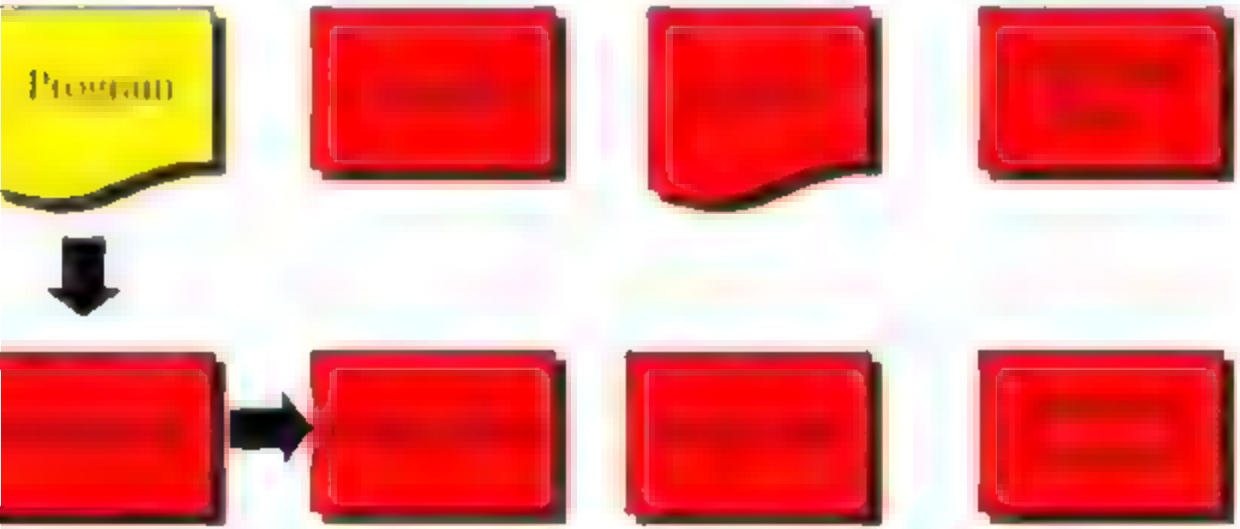
Hot/Cold Structure Splitting Algorithm

- Profile field access counts
- Identify hot structure fields
- Remove cold fields & place in new structure (link from old)
- Insert additional allocation call for new structure
- Access cold fields through new structure

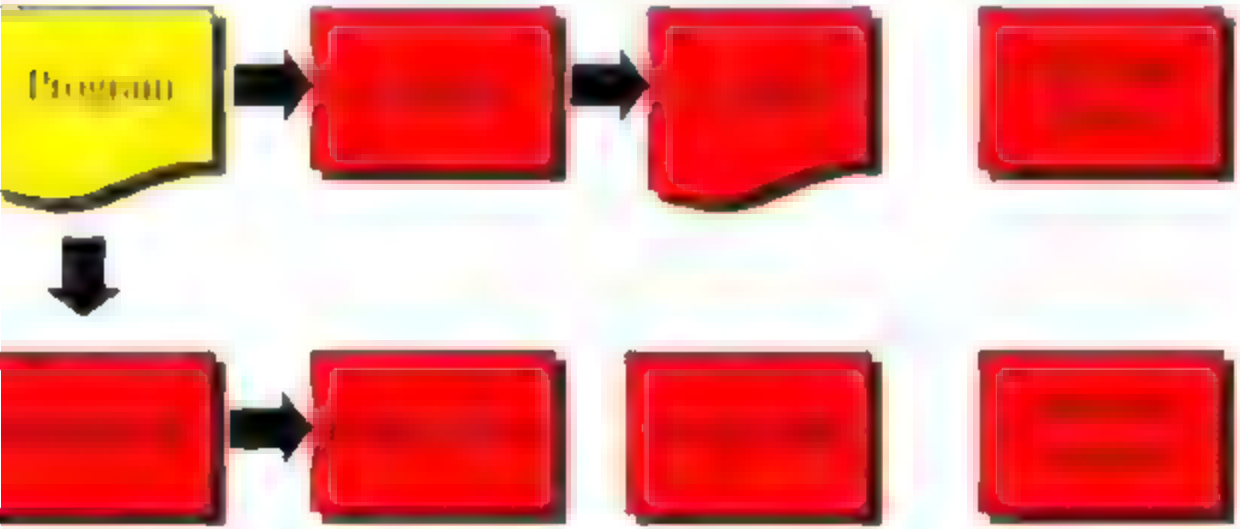
Cache-Conscious Reorganization (with Structure Splitting)



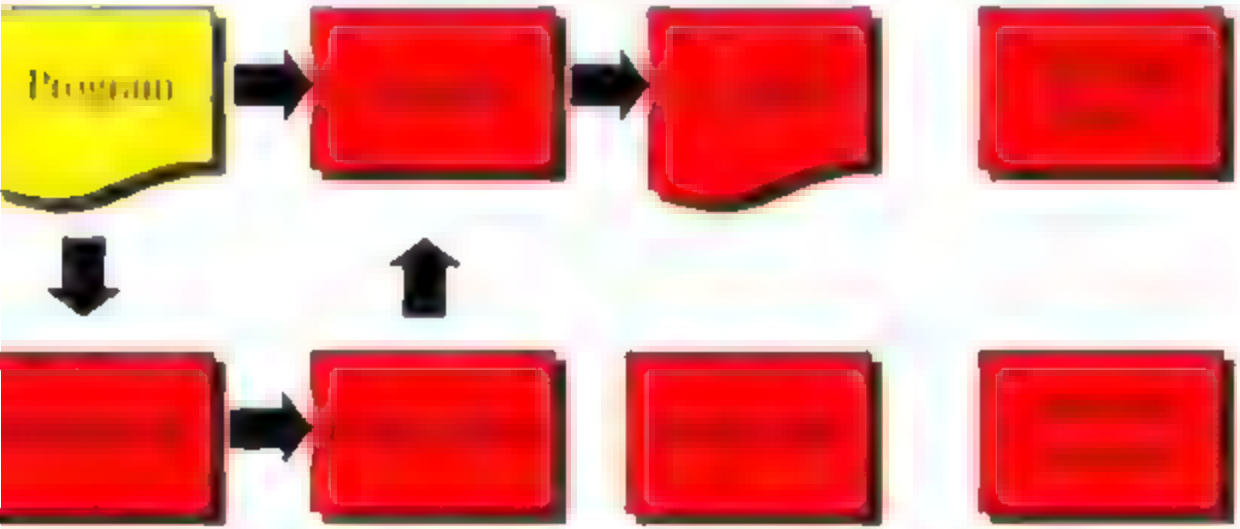
Cache-Conscious Reorganization (with Structure Splitting)



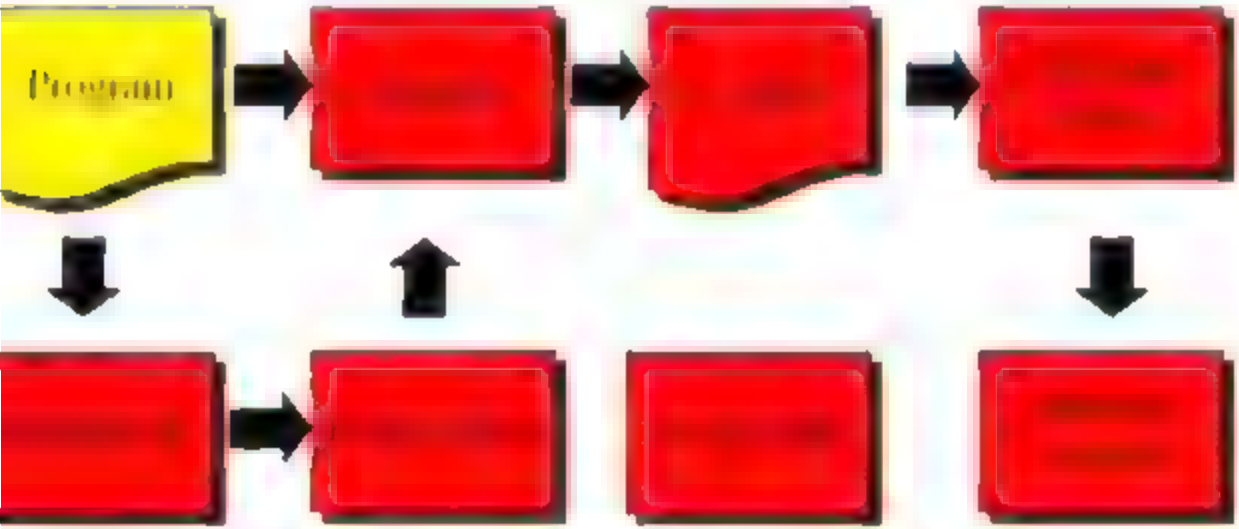
Cache-Conscious Reorganization (with Structure Splitting)



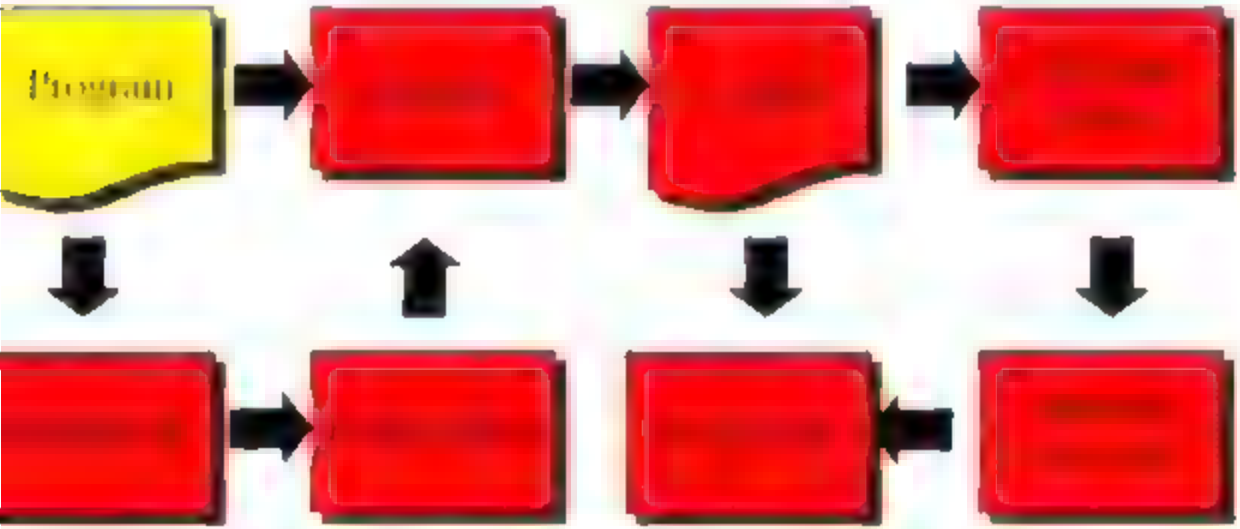
Cache-Conscious Reorganization (with Structure Splitting)



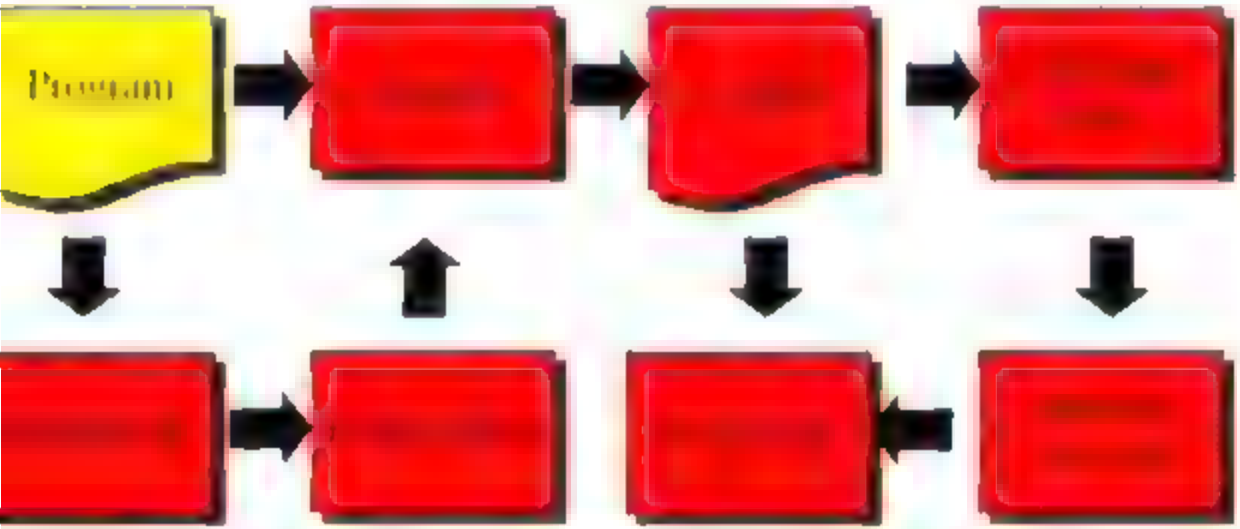
Cache-Conscious Reorganization (with Structure Splitting)



Cache-Conscious Reorganization (with Structure Splitting)

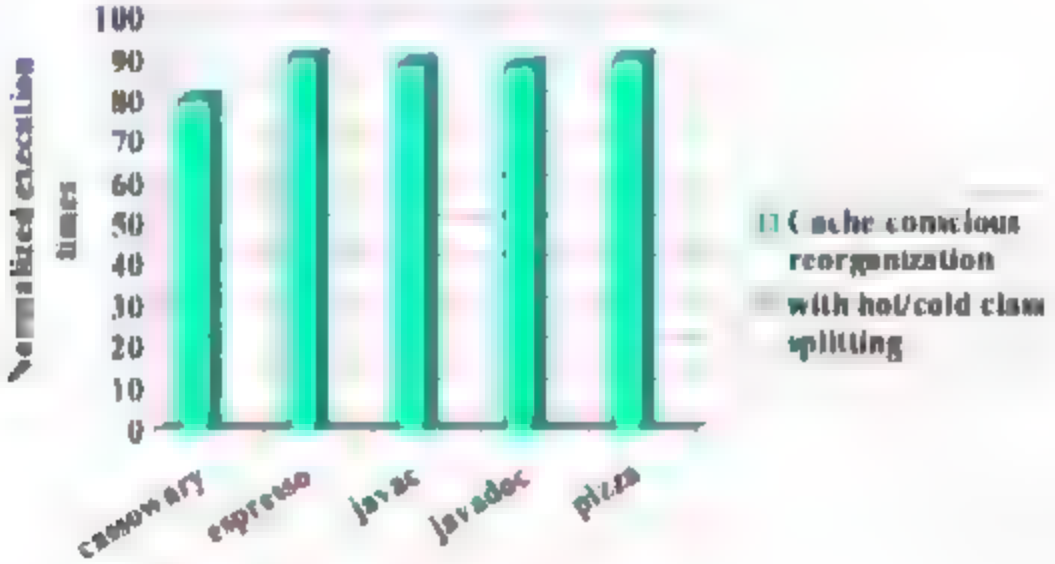


Cache-Conscious Reorganization (with Structure Splitting)

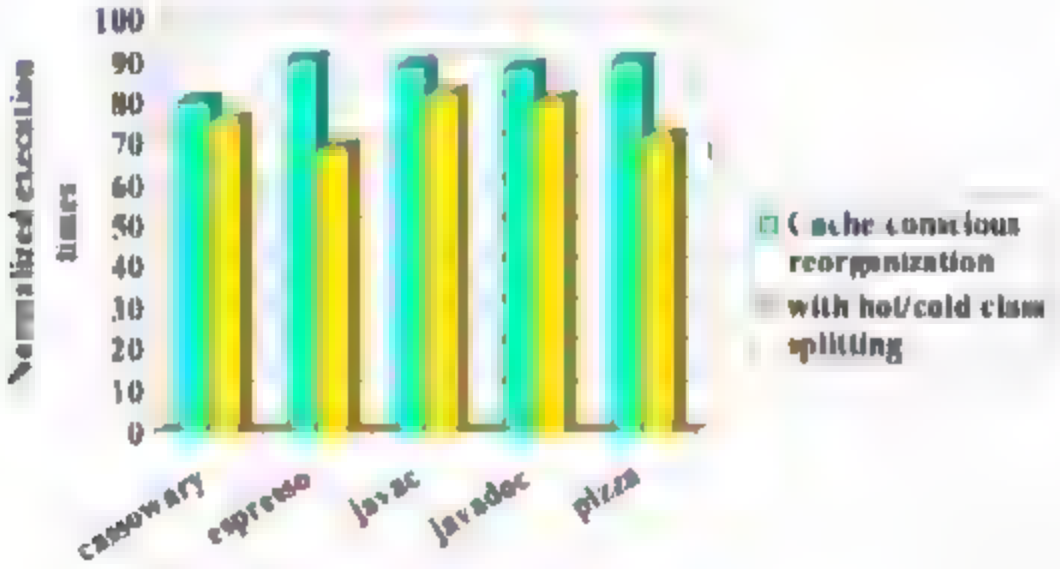


Hot/Cold Class Splitting

Hot/Cold Class Splitting



Hot/Cold Class Splitting



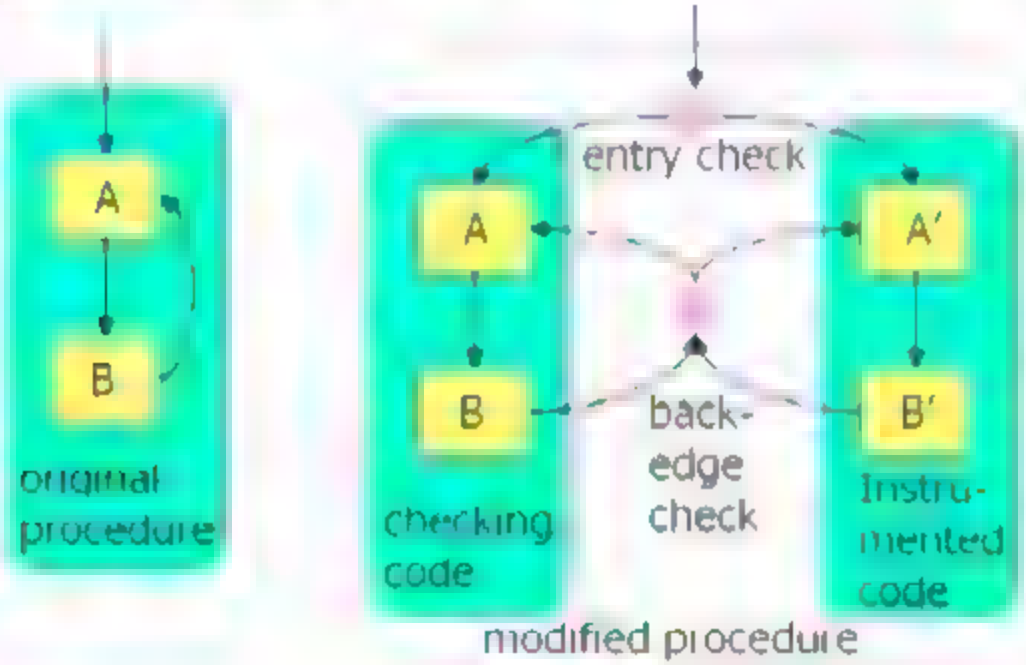
Limitations

- Low-overhead data reference profiling
  - Requires reserving register
  - Potentially inaccurate compiler analysis
- Object Affinity Graph
  - Approximates temporal relationships
  - Higher accuracy increases processing cost

Talk Outline

- Low-overhead data reference profiling
- Fast and accurate data layout determination

Low-overhead data reference profiling



Low-overhead data reference profiling

- 0.1% sampling rate provides high accuracy
- 5% overhead at this sample rate
- Base case is highly optimized x86 code

Talk Outline

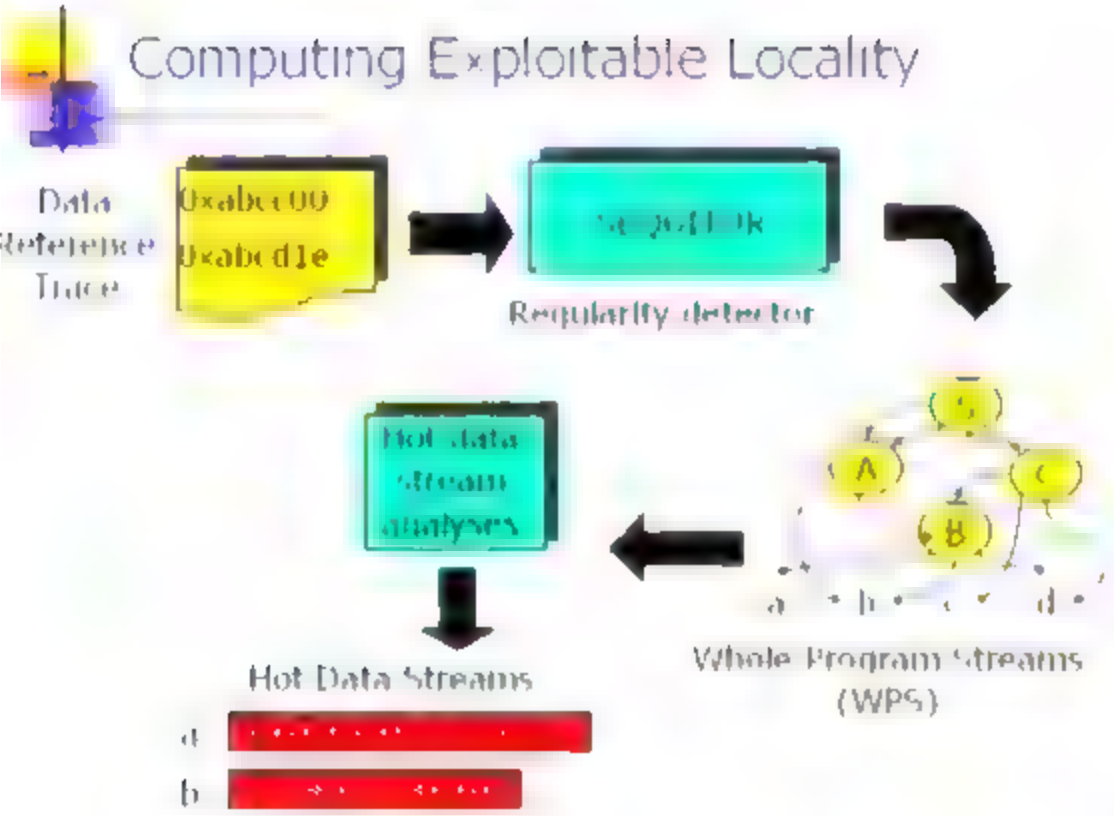
-
-
-
- Fast and accurate data layout determination

Exploitable Locality

- Hot Data Streams:
  Data object sequences that frequently repeat
- Hot=>reference skew + Repetition=>regularity
- Analogous to hot program paths

Hot Data Streams: A B A C E B

Computing Exploitable Locality



Fast, accurate data layout determination

- Fast linear time algorithms
  - Construct Whole Program Stream representation
  - Detect hot data streams
- Few seconds, yet precise information

Conclusions

- Cache-conscious layouts offer large benefits
- GC for cache-conscious data layouts
  - Structure layout reorganization
  - Hot/cold splitting
- Practical within context of CLR
  - Low-overhead data reference profiling
  - Fast, accurate data layout determination
- Opportunity for C# to approach performance of C/C++

Future

- Implement and evaluate in the context of the CLR

References (http://research.microsoft.com/~trishulc)

- GC for cache-conscious data layout
  - Using GC to implement cache-conscious data layout, ISMM 98
  - Cache-conscious structure definition, PLDI 1999
- Low-overhead data reference profiling
  - Bursty tracing: A low-overhead framework for temporal profiling, FDDO 2001 submission
- Fast, accurate data layout determination
  - Efficient data reference locality abstractions and representations, PLDI 2001